Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Geração de números aleatórios e pseudo-aleatórios

Diogo Miguel Gândara Andrade Gomes

Orientador(es) | Carlos Correia Ramos

Irene Pimenta Rodrigues

Évora 2023

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Geração de números aleatórios e pseudo-aleatórios

Diogo Miguel Gândara Andrade Gomes

Orientador(es) | Carlos Correia Ramos

Irene Pimenta Rodrigues

Évora 2023

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Teresa Gonçalves (Universidade de Évora)
Vogais	|	Carlos Correia Ramos (Universidade de Évora) (Orientador) Lígia Maria Ferreira (Universidade de Évora) (Arguente)

Évora 2023

Resumo

Geração de números aleatórios e pseudo-aleatórios

Numa era definida pelo crescimento tecnológico exponencial, as metodologias computacionais sofisticadas tornaram-se parte integrante da exploração e simulação de fenómenos físicos e sociais complexos. O rápido desenvolvimento das tecnologias de informação, juntamente com a proliferação de plataformas centradas em dados, sublinha a necessidade imperativa de geradores de números aleatórios altamente eficientes e seguros.

Este documento fornece uma exploração de várias vias de geração de números aleatórios, englobando as suas diversas classificações, papéis fundamentais e estruturas de teste. No entanto, o ponto crucial deste estudo reside na conceção de um algoritmo baseado nos princípios dos autómatos celulares. O objetivo principal é conceber algoritmos capazes de servir como geradores pseudo-aleatórios, adaptando-se ao cenário em constante evolução das aplicações baseadas em dados.

Palavras-chave: Geradores de Números Pseudo-aleatórios, Autómatos Celulares, Iteradas de Aplicações, Algoritmos, Testes Estatísticos

Abstract

Random and pseudo-random number generation

In an era defined by exponential technological growth, sophisticated computational methodologies have become integral for exploring and simulating intricate physical and social phenomena. The rapid development of information technologies, coupled with the proliferation of data-centric platforms, underscores the imperative for highly efficient and secure random number generators.

This document provides an insightful exploration into various avenues of random number generation, encompassing their diverse classifications, pivotal roles, and rigorous testing frameworks. However, the crux of this study lies in the conception of an advanced generative algorithm based on cellular automata principles. The primary objective is to engineer algorithmic contenders capable of serving as potent Pseudo-Random Generators, seamlessly adapting to the evolving landscape of data-driven applications.

Keywords: Pseudo-random Number Generators, Cellular Automata, Application Iterations, Algorithms, Statistical Tests

Agradecimentos

Em primeiro lugar, gostaria de apresentar os meus sinceros agradecimentos à Universidade de Évora e ao curso de Engenharia Informática, bem como a todo o corpo docente que contribuíram para o enriquecimento do meu conhecimento e me proporcionaram experiências que de outra forma não teria. Apresento um especial agradecimento aos Docentes Carlos Ramos, Irene Rodrigues e Carlos Pampulim Caldeira, por toda a simpatia, disponibilidade e vasto conhecimento que providenciarem na área de matemática, programação e bases de dados.

Um especial obrigado ao professor Carlos Ramos por me ter apresentado e sugerido este tema de dissertação e por toda a disponibilidade, simpatia e paciência que mostrou para comigo ao longo de todo o processo de realização da dissertação.

Não posso deixar de expressar a minha imensa gratidão aos meus pais. São o alicerce da minha vida e da minha educação, e a minha jornada académica não teria sido possível sem o amor, apoio e sacrifício.

Mãe e Pai, desde o momento em que entrei na faculdade, vocês estiveram sempre ao meu lado. Incentivaram-me a perseguir os meus sonhos, mesmo quando as coisas se tornaram difíceis e apoiaram-me mesmo quando parecia impossível. Acreditaram sempre em mim, e isso deu-me a confiança necessária para enfrentar todos os desafios que surgiram no meu caminho.

Quero expressar a minha profunda gratidão à minha incrível namorada de longa data, Margarida Fitas. Durante esta jornada desafiadora que foi o meu mestrado, a Margarida esteve ao meu lado de uma forma que as palavras não conseguem expressar. Ela foi, é e sempre será o meu refúgio em todas as dificuldades que enfrento diariamente.

Além de seres a minha fonte inesgotável de apoio emocional, és a melhor companheira que podia pedir, entusiástica nos meus triunfos e realizações. A tua alegria genuína ao celebrar as minhas vitórias tornou cada conquista ainda mais significativa.

Além disso, a tua capacidade de me inspirar e motivar é algo indescritível. A nossa jornada tem sido e é a coisa mais bonita que me aconteceu, e não tenho dúvidas de que a nossa relação continuará a ser uma fonte constante de força e felicidade em todas as etapas que estão por vir.

"Aos meus queridos pais e à minha incrível namorada Margarida Fitas, esta conquista é fruto do vosso amor, apoio e inspiração. Os meus pais, que foram o alicerce da minha vida, e a minha namorada, que foi o meu porto de abrigo durante esta jornada, merecem o meu mais profundo agradecimento. O vosso papel na minha vida é inestimável, e esta vitória é partilhada convosco. Obrigado por estarem sempre ao meu lado."

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas

Abreviatura/Sigla	**Descrição**
PRNG	*Pseudo Random Number Generator*
TRNG	*True Random Number Generator*
RNG	*Random Number Generator*
CA	*Cellular Automaton*
CSPRBG	*Cryptographically Secure PseudoRandom Bit Generator*
TGFSR	*Twisted Generalized Feedback Shift Register*
NIST	*National Institute of Standards and Technology*
RSA	*Rivest-Shamir-Adleman*
MT	*Mersenne Twister*
LCG	*Linear congruential generator*
LSFR	*Linear Feedback Shift Register*
s.d.	Sem Data

Capítulo 1

Introdução

No cenário tecnológico em constante evolução dos dias de hoje, a geração de números aleatórios desempenha um papel crítico em diversas aplicações, desde simulações científicas até sistemas de segurança cibernética. A criptografia robusta é essencial para garantir a confidencialidade e a integridade das trocas de informação em redes de comunicação. Neste contexto, a geração de sequências aleatórias, como chaves criptográficas, é a base da segurança dos sistemas de comunicação.

Existem diversas formas de criar ou gerar números aleatórios, neste trabalho apresentam-se geradores de números aleatórios baseados em autómatos celulares.

A crescente dependência da sociedade moderna em sistemas digitais, aliada às ameaças constantes à segurança cibernética, ressalta a necessidade de abordagens inovadoras na geração de números aleatórios. Atualmente, os algoritmos para gerar números aleatórios são componentes cruciais em diversas aplicações críticas, incluindo criptografia, simulações computacionais e outras, como jogos online.

O comportamento de certos autómatos celulares permite construir geradores de números aleatórios. Desta forma, é possível criar de modo rápido e eficaz diversos geradores de números aleatórios que podem ser estudados realizando alguns testes para aferir a sua qualidade.

Neste trabalho explora-se a relação entre os autómatos celulares e a geração de números aleatórios. Os autómatos celulares são modelos matemáticos discretos que evoluem em etapas com base em regras simples. A sua capacidade de simular sistemas complexos a partir de princípios fundamentais tem despertado um interesse crescente.

Neste documento, no capítulo 2, apresenta-se uma revisão abrangente da literatura, abordando tanto os princípios dos geradores de números aleatórios, os métodos determinísticos utilizados, e os métodos estatísticos para testá-los. Também se aborda a distinção entre geradores de números pseudo-aleatórios e geradores de números verdadeiramente aleatórios, esclarecendo os desafios de cada um. Além disso, introduz-

se o conceito de autómatos celulares, destacando-se as suas regras fundamentais e o seu potencial para gerar sequências aleatórias.

No capítulo 3, apresenta-se o algoritmo generativo desenvolvido, cujo processo e resultados serão minuciosamente explicados e discutidos nos capítulo 4 e 5. A abordagem utilizada tem potencial para transformar a maneira como é percebida e alcançada a aleatoriedade na geração de números pseudo-aleatórios, através da criação rápida de diversos candidatos a geradores de números aleatórios num curto espaço de tempo.

Finalmente no capítulo 6 apresentam-se as conclusões e perspectivas de trabalho futuro.

Capítulo 2

Revisão de Literatura e Fundamentação Teórica

2.1 Introdução

A geração de números aleatórios é de grande importância na área da tecnologia por diversas razões, sendo que uma das principais encontra-se na simulação em tempo real de fenómenos naturais ou sociais. O poder computacional de hoje em dia permite realizar simulações realistas, não havendo necessidade de se montar experiências caras e de difícil desenho. Nas telecomunicações e processamento de informação os geradores de números aleatórios são imprescindíveis.

Nos dias de hoje o número de pessoas que opta por realizar as suas compras *online* é cada vez maior e, com isto, existe também uma maior oportunidade para que pessoas com intenções maliciosas consigam realizar algum tipo de furto de dados como, por exemplo, os dados do cartão de crédito de uma pessoa. Como tal, é necessário que os sistemas criptográficos e os métodos que são realizados para criar geradores aleatórios estejam à altura desta evolução e que seja possível realizar este tipo de compras sem qualquer tipo de compromisso. Existem outros exemplos onde a realização de métodos robustos de números aleatórios é importante, como *websites* que pedem outro tipo de informações pessoais como a morada.

Existem diversas formas de criar números aleatórios. Em primeiro lugar é necessário separar os conceitos **PRNG**, *Pseudo-Random Number Generator*, de **TRNG**, ou *True Random Number Generator*. Um **PRNG** é um gerador de números pseudo-aleatórios determinístico onde os números se repetem depois de um certo período de tempo e o número que sai como *output* não é, de facto, aleatório. Por outro lado, um **TRNG** é um método que gera números verdadeiramente aleatórios, não é determinístico e não é periódico (Obe (2021)).

Ao longo deste capítulo serão apresentados diversos métodos determinísticos e não determinísticos e serão também estudados os métodos utilizados nas linguagens de

programação Python e C.

2.2 Importância da Aleatoriedade na Criptografia

Normalmente, quando se fala da segurança de sistemas pensa-se num sistema criptográfico com um grande nível de aleatoriedade, uma vez que os valores que saem do mesmo devem ser visualizados por possíveis atacantes como uma sequência de números aleatória, sem revelar qualquer tipo de informações. A aleatoriedade tem uma elevada importância uma vez que muitos dos sistemas criptográficos de hoje em dia têm em conta o *Kerckhoff principle* (Mattioli (2019), Marton, Suciu e Ignat (2010)). Este princípio afirma que, num sistema criptográfico, onde a maior parte das informações são públicas ou fáceis de descobrir, a segurança, em termos criptográficos, deve depender apenas do conhecimento de uma chave (Marton, Suciu e Ignat (2010)). Sendo assim, o que dita se um sistema de segurança é forte é justamente esta chave.

Uma das utilizações mais comuns, e importante no contexto deste documento, é a utilização e geração de números aleatórios para gerar esta chave da forma mais aleatória possível. Um exemplo da utilização e da importância dos números aleatórios é a sua utilização em sistemas simétricos ou assimétricos, onde uma sequência aleatória de números servirá de chave para transformar texto encriptado em texto normal, ou o contrário (Mattioli (2019)).

Para que uma sequência de números seja considerada aleatória, esta deve produzir resultados independentes que devem ser distribuídos de forma uniforme, ter valores imprevisíveis e correlação de dados. Como tal, torna-se importante saber decidir qual o gerador a utilizar, dependendo do contexto, e decidir como testar o mesmo (Marton, Suciu e Ignat (2010)), pois, existem diversos casos onde um gerador passou a um certo número de testes, elevando a confiança posta no mesmo, e acabou por ser possível descobrir a mensagem, ou chave, encriptada. Sendo assim, um dos grandes problemas da aleatoriedade e da segurança de um sistema criptográfico é não existir uma certeza sobre se o que estamos a utilizar é, de facto, seguro.

Normalmente, quanto maior for o número de testes a que um gerador consegue passar, por exemplo os testes **NIST**, maior será a confiança depositada no mesmo. O próprio **NIST** recomenda que cada teste escolhido para ser utilizado, seja aplicado em pelo menos 1000 sequências (Marton, Suciu e Ignat (2010)).

No entanto, depositar demasiada confiança num gerador poderá levar a consequências catastróficas. Na referência de Marton, Suciu e Ignat (2010), é dado um exemplo deste falso sentimento de segurança, com um gerador que passou a um certo número de testes estatísticos e estava a ser considerado bom. A análise ao seu modo de funcionamento revelou que a sua sequência foi baseada nos dígitos de π, que é extremamente previsível. Como tal, a junção de testes estatísticos com uma análise a fundo e a criação de um bom gerador, que é o objetivo deste estudo, será a melhor maneira de ter um gerador considerado seguro.

Existem vários tipos de geradores de números aleatórios. Serve a próxima subsecção para apresentar os mesmos.

2.3 Métodos Determinísticos para Criar Números Pseudo-Aleatórios

Os números pseudo-aleatórios são números gerados de forma determinística, por um computador. Assim, nenhum número obtido através destes geradores é realmente aleatório (Menezes, Van Oorschot e Vanstone (2018)), como referido anteriormente.

Existem, no entanto, algumas formas de ganhar alguma confiança nestes geradores, através da realização de diversos testes estatísticos específicos para tentar encontrar falhas e características comuns das sequências que originam destes geradores (Menezes, Van Oorschot e Vanstone (2018)). No entanto, apesar de ser necessário a verificação destes testes, tal não poderá garantir a segurança total.

Em termos de segurança, e segundo o livro de Menezes, Van Oorschot e Vanstone (2018), o requisito mínimo para que um gerador seja um **CSPRBG** (*Cryptographically Secure PseudoRandom Bit Generator*) é o facto do tamanho k de dada *seed* aleatória necessitar de ser suficientemente grande de modo a que uma pesquisa além de 2^k elementos seja inviável para um *hacker*. Uma das formas de um *hacker* descobrir, por exemplo, uma palavra-passe é utilizando *brute force*, experimentando diversas sequências até chegar a uma conclusão. Se uma sequência for previsível, este processo torna-se relativamente fácil para quem está a realizar o ataque.

De seguida será feita uma pequena apresentação sobre alguns exemplos de métodos para gerar números pseudo-aleatórios e sobre como se pode gerar números pseudo-aleatórios nas linguagens de programação Python e C.

Existem diversos geradores e métodos para gerar números pseudo-aleatórios. Como tal, nas próximas subsecções serão descritos alguns.

2.3.1 *Linear Congruential Generators*

O **LCG** trata-se de um gerador que produz sequências de números pseudo-aleatórias, utilizando uma recorrência linear. Este método é considerado um gerador de números pseudo-aleatórios clássico, sendo estes conhecidos por realizar eficazmente diversas simulações de Monte Carlo (técnica estatística que usa números aleatórios para resolver problemas complexos através de aproximações sucessivas, bastante utilizada em áreas como física, finanças e engenharia), mas bastante fracos no que toca a sistemas de criptografia (Goldwasser e Bellare (1996)).

O **LCG** trata-se de um algoritmo demasiado previsível, uma vez que funciona à base de respostas lineares. Assim, é possível, por exemplo, através do *feedback* que o algoritmo produz, descobrir sequências (e, como tal, descobrir o número que foi gerado) facilmente (Li e Sun (2005)).

Este algoritmo é fortemente desaconselhado para geradores de números pseudo-aleatórios com utilidade prática, sendo possível a sua utilização em contextos restritos ou como peça integrante de outros algoritmos mais sofisticados. No entanto, existem possibilidades de melhorar um pouco o desempenho deste algoritmo.

Sabemos, então, que o grande problema deste algoritmo é a sua previsibilidade e o facto das sequências por ele geradas terem períodos tendencialmente curtos, o que leva a que a sequência de números gerados comece, eventualmente, a repetir-se. Como tal, se o objetivo for tornar o algoritmo viável para utilizar num sistema criptográfico, teremos que arranjar forma de proteger os dados e torná-los o mais imprevisíveis possível. Este caso de previsibilidade é tão grave ao ponto de Krawczyk ter descoberto e desenvolvido um algoritmo de inferência que utilizando as propriedades do **LCG**, consegue prever todas as sequências criadas pelo mesmo (Bellare, Goldwasser e Micciancio (1997)).

Assim, parece improvável arranjar alguma forma de conseguir tornar este algoritmo viável, mas existem algumas formas de contornar os seus problemas.

O único modo de conseguir tornar este algoritmo mais seguro é se for possível gerar a sequência de forma isolada de qualquer outro gerador (Ritter (1991)), ou seja, o algoritmo precisa de operar de maneira independente e não pode ser suscetível a influências externas ou previsíveis. No artigo de Bellare, Goldwasser e Micciancio (1997) foi realizada uma experiência em que utilizaram o algoritmo de inferência referido anteriormente, chamado *Plumsteads's algorithm* (para mais informações sobre o processo, ver (Bellare, Goldwasser e Micciancio (1997)), e foi concluído que o algoritmo é bastante eficaz no que toca a prever o que o **LCG** gera. Foi ainda concluído que é apenas possível ter bons resultados isolando os números criados e que é preciso que o atacante não consiga ver cinco ou mais números consecutivos para que possa ser seguro.

Concluindo, este algoritmo pode sim ser uma opção para um sistema de criptografia mas se e só se for possível arranjar uma forma eficaz de isolar e esconder os números pseudo-aleatórios que serão criados e gerados pelo **LCG**. Dependente do contexto, no entanto poderá ser mais fácil e, mais importante, mais seguro optar por outro algoritmo onde não seja necessário gastar tempo e recursos para torná-lo mais seguro.

2.3.2 RSA

Este método, ao contrário do anterior, é conhecido por ser um **CSPRBG** (Menezes, Van Oorschot e Vanstone (2018)), ou seja, é um algoritmo que pode ser utilizado na criptografia devido à sua segurança e implementação.

O **RSA** é considerado o primeiro sistema de criptografia assimétrico onde a sua segurança depende do IFP (*Integer Fatorization Problem*) (Mahto, Khan e Yadav (2016)). O **RSA** funciona utilizando um par de chaves pública e privada. A chave pública (n, e) pode ser visualizada e partilhada com qualquer pessoa e a privada (p, q, d) é secreta e apenas visível e conhecida pelo proprietário. O valor n representa

o produto de dois números primos p e q, o valor e é o expoente público ou de criptografia e o valor d é o expoente privado que é utilizado para decifrar a mensagem privada utilizando a chave pública correspondente e.

É utilizada a chave pública do destinatário para criptografar uma mensagem e o destinário só consegue abrir a mesma se e só se tiver a chave privada (Menezes, Van Oorschot e Vanstone (2018)).

Um dos aspetos mais importantes a ter em conta sobre um sistema de criptografia é tornar as chaves, processos, entre outros, o mais forte possível, tornar imprevisível o comportamento dos algoritmos escolhidos e chegar a uma sequência que dá, por exemplo, acesso a uma mensagem secreta.

No artigo de Mahto, Khan e Yadav (2016), foi utilizado o algoritmo em questão com o **CCG** (*Cubic Congruential Generator*), para gerar os números primos que são utilizados posteriormente na implementação para depois criar o resto das chaves.

Neste artigo foram também realizado estudos com estes dois algoritmos em conjunto, tendo sido atingidos valores interessantes e, mais importante, eficazes em termos de segurança. Foi provado que a seleção dos números primos utilizados no **RSA** é crucial para um resultado eficaz e seguro (Mahto, Khan e Yadav (2016)), uma vez que se forem escolhidos dois números primos baixos (neste caso 13 e 11) a robustez é baixa e são precisas apenas 19 horas para descobrir a sequência. No entanto, dando um valor baixo (19) e outro valor mais alto, mas médio, (83) demoraria 27 quatriliões de anos a descobrir a sequência.

Mesmo não utilizando um valor baixo e outro médio, por exemplo $p = 503$ e $q = 541$, demoraria 4 triliões de anos a descobrir a sequência (Mahto, Khan e Yadav (2016)). Através deste artigo, podemos perceber que, se escolhidos os números primos com alguma cautela (podendo utilizar outro gerador de números pseudo-aleatórios para o efeito), estamos perante um algoritmo que fornece uma segurança bastante eficaz.

2.3.3 Mersenne Twister

O algoritmo ***Mersenne Twister*** (ou **MT**) é um gerador de números pseudo-aleatórios que é utilizado pela biblioteca de python, chamada random.py. No entanto, ao contrário do **RSA** e em semelhança ao **LCG** este gerador não é adequado para um sistema de criptografia.

Este é um algoritmo que originou com base no **TGFSR**, ou ***T**wisted **G**eneralized **F**eedback **S**hift **R**egister*, que tem um período de $2^{19937} - 1$. Este algoritmo, apesar de ter diversas características que o tornam num bom PRNG, para algumas situações, não é adequado, como já referido, para sistemas de criptografia.

Tal como o **LCG**, o grande problema do **MT**consiste no facto de a sequência que é gerada por este ser demasiado previsível, uma vez que existe facilidade em prever o estado seguinte devido aos *outputs* dados pelo **MT**, devido à sua linearidade e ao grande *output* de 19937 *bits* (Jagannatam (2008)). Segundo este mesmo artigo, uma

das formas que os autores do **MT** recomendam para que seja possível utilizar este algoritmo em sistemas de criptografia seria esconder os *outputs* através de funções de *hash*.

2.3.4 CryptMT

O **CryptMT** seria um **MT** modificado de modo a que seja seguro o suficiente para poder utilizar num sistema de criptografia. Existem duas versões de **CryptMT**, a primeira e a mais recente (para mais informações sobre a implementação destes, ver a referência de Jagannatam (2008)):

- **Versão 1**: Utiliza o MT como o gerador base e combina um filtro multiplicativo com 32 *bits* de memória e, por fim, calcula o produto cumulativo do que vem como *output* do **MT** e utiliza os 8 *bits* considerados mais importantes. Além disso, ainda é utilizado um conjunto diferente de parâmetros de inicialização para melhorar a qualidade dos números que são gerados.
- **Versão 3**: Esta versão tem um período maior que as outras versões (2^{251}), o que levará a que seja possível gerar uma sequência de números pseudo-aleatórios muito maior antes que exista repetição. Esta versão é melhor que a anterior, uma vez que tem 128 *bits* e é mais rápida e segura.

Concluindo, se utilizado o **CryptMT** em vez do **MT** existe a possibilidade de se ter um sistema de criptografia eficaz e seguro. No entanto, é necessário realizar mais testes para avaliar se realmente este seria melhor que outros candidatos (Jagannatam (2008)).

2.3.5 Monte Carlo *Algorithms*

Ao contrário dos anteriores, o nome ***Monte Carlo*** refere-se não apenas a um algoritmo, mas sim ao conjunto de algoritmos de um tipo de algoritmos probabilísticos - polinomiais (Goldwasser e Bellare (1996)). É dado o nome de polinomiais a este tipo de algoritmos uma vez que terminam com um número de passos polinomial e com um *output* correto se e só se existir uma grande probabilidade associada(Goldwasser e Bellare (1996)).

Estes algoritmos têm características e comportamentos únicos, sendo que uma simulação de ***Monte Carlo*** prevê um conjunto de resultados com base num *range* de varlores *versus* um conjunto de valores de *input* fixos (IBM (s.d.)). Assim, devido à sua grande precisão, são utilizados para previsões de longo prazo.

Um dos fatores mais importantes e/ou interessantes neste tipo de algoritmos é o facto de necessitarem de números aleatórios de alta qualidade, ou seja, é necessário que sejam o mais imprevisíveis possível e, ainda, que não existam, ou que existam apenas passado bastante tempo, padrões e sequências de números (Raeside (1976)), sendo que para que um número aleatório seja de alta qualidade necessita de passar no maior número de testes sofisticados possível. A partir de números aleatórios de

alta qualidade, o algoritmo gera várias amostras que são posteriormente utilizadas para calcular uma estimativa da solução com uma média dos resultados.

Sendo assim, parece que este tipo de algoritmos são promissores para o estudo do presente documento, uma vez que, tendo em conta as suas características, a sua utilização poderá gerar números aleatórios de forma mais consistente que outros algoritmos e métodos determinísticos (uma vez que o *output* do ***Monte Carlo*** é probabilístico e dependente da qualidade dos números aleatórios utilizados). No entanto, estes algoritmos não são, normalmente, utilizados para criptografia mas sim para simulações (inclusive, existem muitos algoritmos que são utilizados para fazer simulações de ***Monte Carlo***).

Segundo a referência de IBM (s.d.), existem três passos a serem seguidos:

- Identificar as variáveis necessárias - a que será a prever (que é a dependente) e as variáveis independentes, de modo a configurar o modelo.
- Especificar e estudar as opções para as distribuições de probabilidade para cada uma das variáveis independentes.
- Realizar e repetir diversas simulações que criam valores aleatórios através das variáveis independentes. Este último passo é feito até que existam resultados suficientes para obter uma amostra que represente um número quase infinito de combinações possíveis do contexto presente.

2.3.6 Nas Linguagens Python e em C

Agora, será interessante rever o que existe sobre estes métodos em algumas linguagens, como o Python e o C, e perceber o quão seguros são os métodos utilizados nestas. Em python existem algumas bibliotecas de interesse para a geração de números pseudo-aleatórios, tais como a biblioteca random.py (ver (*random — Generate pseudo-random numbers — Python 3.11.0 documentation* (s.d.)). Esta biblioteca implementa geradores de números pseudo-aleatórios para várias distribuições matemáticas (*random — Generate pseudo-random numbers — Python 3.11.0 documentation* (s.d.)).

À primeira vista, para qualquer pessoa que já tenha utilizado esta biblioteca de Python, pode parecer que se trata de uma ferramenta, ou forma, segura de gerar números aleatórios. No entanto, este não é o caso.

Esta biblioteca de Python utiliza um gerador pseudo-aleatório de números chamado **Mersenne Twister** para gerar os números aleatórios e, apesar da implementação em si, ser eficaz e segura, esta não serve para utilizar quando o objetivo é ter um sistema criptográfico seguro (*random — Generate pseudo-random numbers — Python 3.11.0 documentation* (s.d.)). Sendo assim, esta biblioteca é considerada eficaz quando o objetivo é apenas gerar números pseudo-aleatórios para um objetivo trivial e nunca deve ser utilizado para fins de segurança, como criptografar palavras-passe, por exemplo.

No entanto, existe outro módulo que pode ser utilizado, o ***secrets module*** que a própria documentação do Python recomenda (*random — Generate pseudo-random numbers — Python 3.11.0 documentation* (s.d.)). O ***secrets module*** é utilizado para gerar de forma criptográfica números aleatórios fortes que podem ser utilizados para fins de segurança, como palavras-passe, autenticações, entre outros (*random — Generate pseudo-random numbers — Python 3.11.0 documentation* (s.d.)).

Sendo assim, o módulo ***random*** cria números pseudo-aleatórios através de uma *seed* do computador, gerando uma sequência pseudo-aleatória de números. Se um *hacker* descobrir esta *seed* terá facilmente acesso à sequência através de algo tão trivial como *brute force* (experimentar diversas sequências até chegar à correta) (Sinha (2021)). Já o módulo ***secrets*** funciona através das funções que geram tokens totalmente aleatórios que podem ser guardados em forma de bytes, texto, entre outros (*PEP 506 – Adding A Secrets Module To The Standard Library | peps.python.org* (s.d.)), tornando os dados criptografados, através deste módulo, fortemente aleatórios e imprevisíveis (Arun (2022)).

Em linguagem C, existe uma função de random chamada rand. Esta gera um número pseudo-aleatório e, tal como a função do Python, também não é segura para utilizar em sistemas de criptografia (T. (2022)).

No entanto existe a função rand_s que é mais segura que a anterior (T. (2022)). Tal função é considerada segura para utilizar em sistemas de criptografia, uma vez que, assim como o módulo ***secrets***, não utiliza a *seed* mas sim o sistema operativo para gerar números pseudo-aleatórios fortes (T. (2022)).

2.4 Algoritmos Caóticos

Algoritmos caóticos, ou aplicações caóticas, são funções matemáticas que geram padrões complexos com base no valor inicial da *seed* (Naik e Singh (2022)). Embora produzam um estado desordenado e irregular, parecem ser opções promissoras para gerar sequências pseudo-aleatórias seguras.

De acordo com Wang e Cheng (2019), as aplicações caóticas possuem características que as tornam eficazes para sistemas criptográficos. Vários sistemas caóticos demonstraram ser eficientes neste contexto, alcançando um nível de "aleatoriedade perfeita" em algumas demonstrações, aprovando os testes estatísticos mencionados anteriormente.

2.5 Revisão dos Métodos de Testagem de Geradores de Números

A testagem dos geradores de números aleatórios é um passo importante quando se considera a utilização de números aleatórios. Isto, pois, estes métodos são **PRNGs** que são determinísticos e periódicos, o que torna importante perceber se estas duas limitações tornam o **PRNG** não seguro.

Para que um número, ou uma sequência de números, seja aleatório este tem que estar distribuído uniformemente e cada número é independente dos números gerados antes deste (DiCarlo (2012)). Um gerador de números aleatórios pode ser definido por um sistema que cria sequências aleatórias de números aleatórios. Dependendo do contexto, na área da criptografia escolher um gerador que não seja totalmente seguro pode levar a grandes consequências.

Sendo assim, podemos estudar e perceber os tipos de geradores que existem e ainda o processo de decisão dos mesmos.

2.6 Geradores de Números Aleatórios

Nesta seccção, serão apresentados os **PRNGs**, que são a base do estudo, e os **TRNGs**.

2.6.1 Geradores de Números Pseudo-Aleatórios

Os **PRNGs** não conseguem gerar números totalmente aleatórios, uma vez que são determinísticos. Este tipo de gerador cria a sua própria entropia de modo a tornar-se o mais próximo de aleatório possível, enquanto que os **TRNGs** utilizam fenómenos físicos, por exemplo, para criar esta entropia. Assim, os **PRNG** utilizam recursos do computador, como o relógio (utilizando a data e hora) e tentam criar sequências aleatórias através destas (DiCarlo (2012)). O que determina o quão imprevisível será o resultado determinista deste tipo de *Random Number Generators* é o quão imprevisível é a *seed*.

É percetível que se torna perigoso utilizar este tipo de geradores se for de fácil acesso, a um atacante, a data e hora do computador, *seed*, entre outros, sendo normalmente apenas necessária *brute force*. Um exemplo onde isto é um problema, é a função *random* do Python que utiliza a *seed* do computador para gerar sequências "aleatórias". Como referido anteriormente, se o objetivo for algo trivial, como criar números aleatórios de 1 a 100 para saber que lugares são atribuídos a um conjunto de pessoas (por exemplo), este **RNG** é eficaz, caso contrário não é boa prática utilizá-lo.

Para além dos testes que são feitos, os recursos de entropia de uma dada máquina, o difícil acesso aos valores iniciais do gerador e a impossibilidade de prever estados futuros a partir de estados anteriores terão um papel fundamental para determinar se um **PRNG** é forte ou não (DiCarlo (2012)). Em casos onde não exista a necessidade

de utilizar aleatoriedade perfeita, estes são a escolha clara, uma vez que são mais eficientes, têm menos custos e produzem *outputs* grandes em pouco tempo (Mattioli (2019)). Existem diversos estudos ativos que têm como objetivo encontrar formas seguras de criar **PRNGs** seguros, levando a que existam os **CSPRNG**, sendo um exemplo o algoritmo RSA.

Outra característica dos **PRNGs** é a sua periodicidade. Independentemente dos **PRNGs** serem mais ou menos sofisticados, existirá sempre um momento em que existe repetição de sequências. Logo, quanto maior for o seu período, maior será a qualidade do mesmo.

2.6.2 Geradores de Números Aleatórios Verdadeiros

Este tipo de gerador de números aleatórios combate os problemas e limitações que o anterior tem, através da utilização de meios físicos como o nível da radiação num certo sítio a uma certa hora, uma vez que não é determinístico nem periódico. Contudo, este também tem as suas próprias limitações.

Apesar de apresentarem um alto nível de imprevisibilidade, por vezes não apresentam uma distribuição uniforme, podendo existir tendência para sair mais vezes um certo valor ou existindo algum tipo de manipulação do fenómeno físico, o que leva a que não se possa utilizar qualquer fonte física sem estudar as consequências primeiro (Marton, Suciu e Ignat (2010)).

Nos **PRNGs**, percebemos que é necessário criar entropia através de recursos existentes no *host*. Nos **TRNGs** é possível criar ou ir buscar esta entropia a recursos externos, por exemplo físicos. Este tipo de fenómenos físicos têm uma aleatoriedade (quase) perfeita e caso tenham sido aplicados filtros de modo a não existirem casos tendenciosos, torna-se difícil para um atacante descobrir as sequências que serão geradas.

No entanto, este tipo de gerador pode trazer problemas (DiCarlo (2012)):

- O primeiro é o facto de ter um custo muito mais elevado que os anteriores, uma vez que, para além de existir a necessidade de existir uma máquina a realizar o gerador propriamente dito, terá que existir ainda algum tipo de *hardware* que consiga perceber e registar os *outputs* constantes dos fenómenos físicos.
- São um pouco limitados no que toca à produção de números aleatórios e ainda existe a possibilidade de estes *hardwares* externos falharem ou terem defeitos ao longo do tempo.

Um exemplo conhecido deste tipo de gerador é o site Random.org. O qual utiliza ruído atmosférico para gerar o número aleatório, no entanto, uma vez que está *online*, não é apropriado para contextos de segurança já que pode ser intercetado por um atacante (DiCarlo (2012)).

2.6.3 Geradores híbridos

Existem, ainda, alguns casos onde se juntam o melhor dos dois mundos, ou seja, a aleatoriedade dos **TRNGs** com a eficácia e rapidez dos **PRNGs**. Para isso, basta gerar uma sequência aleatória através de um fenómeno físico e, em vez de dar a *seed* como input dá-se o resultado do fenómeno físico ao **PRNG** (Mattioli (2019)). No entanto, este também poderá ter custos elevados e obriga a que se possa confiar no fenómeno físico a todo o momento e a que o **PRNG** esteja bem estruturado.

A adoção do modelo híbrido poderá ser o futuro da geração de números aleatórios. A junção destes dois conceitos num sistema totalmente funcional seria a forma mais eficaz e segura de gerar chaves para sistema criptográficos.

2.6.4 Geradores de números pseudo-aleatórios criptograficamente seguros

Apesar de a utilização de **PRNGs** ser desaconselhada quando o tema é um sistema criptográfico, existe um tipo de **PRNGs** que pode, eventualmente, ser seguro para utilizar, os *Cryptographically Secure Pseudo-Random Number Generators*, ou **CSPRNGs**.

Os critérios para que um **PRNG** possa ser considerado um **CSPRNG** são exigentes e grande parte dos candidatos falha. A única forma de poderem ser considerados seguros é passar no teste *next-bit*, que será mencionado na secção sobre testes, e se nenhum atacante conseguir descobrir e prever os próximos estados através dos anteriores (Mattioli (2019)).

2.7 Chaves Criptográficas

Voltando ao conceito de chaves criptográficas, conseguimos perceber que a obtenção de boas chaves criptográficas é um aspeto com uma importância considerável no âmbito da criptografia. As chaves podem ser consideradas como o coração da segurança dos sistemas, em alguns casos. Recapitulando, estas chaves transformam a informação que está encriptada em texto normal, ou vice-versa. Como tal, estas chaves devem ter um nível de aleatoriedade o mais elevado possível.

A entropia presente num **RNG** é um dos aspetos que dita a qualidade da sequência gerada. A entropia é o que desordena e traz confusão a um sistema. No entanto, a sequência precisa de seguir uma distribuição uniforme e existe uma maior probabilidade desta ser imprevisível caso exista uma entropia elevada. Da mesma forma, se uma sequência de números seguir uma distribuição uniforme mas tiver uma entropia fraca, toda a sequência não poderá ser considerada segura.

Sendo assim, é importante que exista o maior número de condições presentes para que uma sequência seja o mais aleatória possível, algo que nem sempre é de escolha fácil, podendo ter de se escolher uma maior existência de um aspeto em prol de

outro. Trata-se de encontrar o balanço, em busca de produzir a chave criptográfica mais segura possível. Outro aspeto importante é que um sistema criptográfico seguro não deve ter a mesma chave a todo o momento (Mattioli (2019)).

O algoritmo a ser desenvolvido irá criar sequências aleatórias, que intersetam a entropia da natureza dos autómatos celulares com a necessidade de serem uniformemente distríbuidos, podendo ser considerados candidatos a **PRNGs**. Assim, tenta-se encontrar um equilíbrio entre uma entropia forte e imprevisibilidade em conjunto com uma distribuição uniforme, criando sequências que poderão ser, eventualmente, consideradas seguras se o **PRNG** passar em diversos testes.

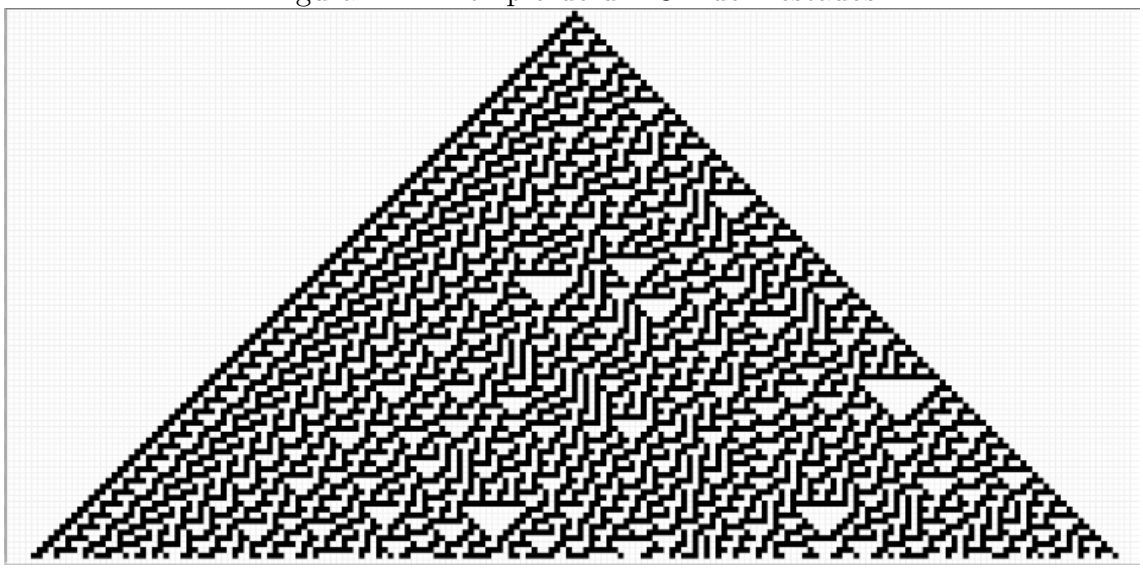
2.8 Autómatos Celulares

Nesta secção serão apresentados os autómatos celulares, as suas características, origens e ainda um pouco da formulação teórica. A aplicação teórica dos autómatos celulares será aprofundada numa secção seguinte, perto da explicação do processo do algoritmo que vai ser criado. Os autómatos celulares são modelos matemáticos que se baseiam numa rede de sítios (células) discretos e idênticos, onde cada sítio pode assumir um conjunto específico de valores inteiros (Wolfram (1983)). Este conceito teve, como origem, um autómato celular auto-replicante, criado por Von Neumann, relacionado com o desenvolvimento de um organismo (Carlos Ramos (2009)).

Estes modelos representam uma abordagem teórica para simular sistemas naturais complexos e são capazes de reproduzir comportamentos observados em sistemas reais. Os autómatos celulares são comparáveis a computadores de processamento paralelo simples devido à capacidade de realizar múltiplos cálculos simultaneamente (Wolfram (1983)). Podem também ser caracterizados pela rede de células, vizinhança entre células, número de estados possíveis de cada célula e regras de transição, sendo que cada autómato tem as suas características únicas, a sua auto-organização e a sua simplicidade (Wolfram (1983)).

Um autómato celular, habitualmente chamados de CA, é constituído por uma rede de células que evoluem ao longo de tempo, de forma sincronizada, tendo em conta a mesma regra local que lhes é dada, conceito que será explicado brevemente, onde cada célula evolui tendo em conta o seu valor e os valores da sua vizinhança (células à direita e à esquerda). A dimensão de um CA depende do número de estados que o mesmo tem e a escolha desta depende do contexto e necessidades, sendo que no presente estudo o foco cairá sobre os autómatos celulares binários que têm 2 estados, $\{0, 1\}$ (Kunkle (2003)). Cada estado ou configuração é apresentada em baixo da anterior, criando gráficos de 2 dimensões, onde as células em branco apresentam-se quando o valor da regra equivale a 0, e as células em preto apresentam-se quando o valor da regra equivale a 1, como podemos ver na figura 2.1.

Figura 2.1: Exemplo de um CA de 2 estados



2.8.1 Categorias de um CA

Os autómatos celulares têm diversas formas de serem utilizados. Segundo Sarkar e outros autores como McIntosh (Kunkle (2003)), podem ser divididos em três categorias. São estas clássica, jogos e moderna. A forma clássica foca-se na pesquisa que deu origem aos CAs, realizada por Neumann. Foi criado um CA com 2 dimensões, 29 estados, uma vizinhança igual a 5 e, no fim, o CA deu origem a um computador universal que conseguia constituir uma máquina através de uma descrição. Como referido anteriormente, este autómato era auto-replicante dando uma descrição a si mesmo e criando novos elementos .

No que toca à categoria de jogos, existe um jogo conhecido no tema dos CAs, chamado Jogo da Vida por Conway, ou *Conway's Game of Life*, criado por John Horton Conway nos anos 60. Este jogo ocorre em duas dimensões, onde cada quadrado de uma matriz bidimensional pode conter uma célula e tem em conta os 8 vizinhos de uma célula. Pode ser jogado diretamente num navegador *web* (Adamatzky (2010)). O jogo está disponível no site https://playgameoflife.com/, que também oferece uma explicação detalhada do seu funcionamento.

No jogo da vida, existem algumas regras (Adamatzky (2010)):

- Cada uma das células tem um estado de viva ou morta (pode ter apenas estes dois estados) tendo a cor preta ou branca, respetivamente, sendo que este estado é constantemente atualizado a cada *step* do jogo e em simultâneo com todas as outras células.
- Uma célula que esteja morta pode voltar à vida sempre que tenha três células vizinhas.

- Uma célula que tenha apenas um ou nenhum vizinho morre, como se fosse por solidão.
- Uma célula com quatro ou mais vizinhos morre, como se fosse para combater a sobre-população.
- Uma célula que tenha dois ou três vizinhos mantém-se viva.

Embora as regras sejam simples, este jogo já exibiu comportamentos complexos, como padrões periódicos, naves espaciais e ainda formas de vida auto-replicantes. Existem alguns estudos realizados para perceber se este jogo pode ser considerado um computador universal e alguns destes conseguiram provar que sim (Kunkle (2003)).

Por fim, temos a última categoria, a moderna, que se trata da classificação dos autómatos celulares, normalmente associada às classes de Wolfram e de Li-Packard.

A primeira é dividida em 4 classes: uniformidade, repetição, aleatoriedade e complexidade (Shiffman (s.d.)):

- Classe 1, Uniformidade - Após um pequeno número de gerações todas as células são constantes. Ou seja, a partir de uma certa altura o autómato celular é igual em todas as próximas gerações.
- Classe 2 , Repetição - Parecida à anterior, esta classe tem uma repetição contínua a partir de uma certa geração, no entanto cria um padrão (por exemplo oscilação entre 0 e 1) e é este que se repete, ao contrário da anterior que fica igual em todo o seu tempo de vida.
- Classe 3, Aleatoriedade - Nesta não existe repetição como nas anteriores, sendo mais difícil de prever e agindo de forma caótica.
- Classe 4, Complexidade - Trata-se de a junção da classe 2 com a classe 3. Existe repetição de padrões, tal como na classe 2, mas estes padrões aparecem de forma aleatória e imprevisível.

A segunda, desenvolvida por Li e Packard, é baseada na classificação anterior, e tem as seguintes classes (Kunkle (2003)):

- *Null*: A configuração limite, que se trata do estado final depois de um certo número de passos, é homogéneo. Esta é semelhante à primeira classe da classificação anterior.
- Ponto fixo: A configuração limite é invariante depois de aplicada a regra apenas uma vez, excluindo regras que caiam na classe anterior. Esta é semelhante à segunda classe da classificação anterior.
- Dois ciclos: A configuração limite é invariante depois de aplicada a regra de atualização 2 vezes.
- Periódico: A configuração limite é invariante depois de aplicada a regra de atualização L vezes, com o tamanho L independente ou quase independente.

- Complexo: Pode ter configurações limite periódicos mas demora muito tempo a chegar à condição limite. O tempo aumenta linearmente com o número de células. Esta é semelhante à terceira classe da classificação anterior.
- Caótico: Por fim, tal como na classificação anterior, a última classificação é caótica. Nesta existem dinâmicas não periódicas caracterizadas pela sua divergência exponencial do tamanho do ciclo da configuração limite com o número de células e pela instabilidade em respeito a perturbações da condição inicial.

No fundo, Li-Packard divide as classificações de Wolfram em três novas, o ponto fixo, dois ciclos e periódica.

2.8.2 Auto-organização

Segundo Wolfram, os autómatos celulares, apresentam uma forma de se auto-organizarem, através de uma equação e de um estado inicial desordenado (Wolfram (1983)). Seria de esperar que se o ponto de partida fosse totalmente desorganizado, então os autómatos não teriam uma capacidade de organização suficiente de forma a existir uma frequência de padrões consistente. No entanto, no artigo referido, foi feita uma experiência, onde os valores atribuídos a cada sítio tinham uma probabilidade igual de ser ou 0 ou 1 onde, mesmo com a aleatoriedade e falta de organização inicial palpável, os autómatos criaram algo totalmente aleatório no início mas, ao longo da evolução das células, passou a ser possível visualizar alguns padrões na forma de triângulos. Esta caraterística dos autómatos celulares torna-os particularmente interessantes, não apenas para o âmbito deste documento, mas também para o campo da segurança e criptografia de forma global.

Este padrão ocorre, uma vez que as células ou cada sítio interage com outras adjacentes e, através destas interações, é criada uma estrutura organizada ao longo do tempo e evolução. Os autómatos celulares têm, portanto, implicações em diversas áreas, nomeadamente na biologia, física, matemática e para a área de segurança, criptografia e números aleatórios.

Depois de estudados os aspetos importantes dos autómatos, serão apresentadas as definições necessárias para a criação de autómatos celulares, em busca da aleatoriedade.

2.8.3 Definições básicas de um Autómato Celular

De modo a utilizar um CA é importante ter em conta três propriedades importantes (Kunkle (2003)):

1. Dimensão, normalmente associada à letra d. Esta dita como cada célula se organiza.
2. Número de estados por célula k, onde $k \geq 2$ que determina o número de valores que uma célula pode ter.

3. Raio, r, que determina o número de vizinhos que tem à direita e à esquerda.

De forma a calcular o estado de uma célula i a um certo tempo t calcula-se S_i^t, considerando-se os $2r + 1$ vizinhos do instante anterior. Na secção de conceitos será explorado mais a fundo os conceitos teóricos de um CA.

2.8.4 Regras Utilizadas nos Autómatos Celulares

Serão agora apresentadas algumas das regras utilizadas em autómatos celulares. O objetivo desta secção será estudar as regras em si mas, principalmente, estudar a sua aplicabilidade em **PRNGs**.

De entre as regras existentes, ou *code rules*, que são possíveis de utilizar nos autómatos celulares, não é por acaso que a primeira a ser apresentada seja a número 30, uma vez que esta é uma das regras que é normalmente ligada à aleatoriedade. Existem diversos testes feitos a algumas *code rules*, ou regras, e a regra 30 é a que costuma dar resultados com maior interesse (Bhattacharjee e Das (2022)).

Stephen Wolfram, autor de alguns artigos e referências do presente documento, defendeu o uso desta regra para a criptografia e para a geração de números pseudo-aleatórios (Gage, Laub e McGarry (2005)).

O conceito de regras, foi elaborado através das possíveis progressões de 3 células ($2^3 = 8$), ou seja, a célula principal, as células à sua esquerda e as células à sua direita. Estas progressões dão como resultado único uma nova célula (Gage, Laub e McGarry (2005)).

Assim, como cada progressão é composta por 3 células, onde cada uma destas tem o valor 0 ou 1 (quando é atribuído o valor 1 a uma célula significa que existirá uma cor preta no lugar da mesma), estamos perante uma sequência de 8 números. No exemplo da regra 30, as sequências são organizadas de 111 até 000, de forma descendente: 111 110 101 100 011 010 001 000.

Na regra 30, a decisão dos *bits* é realizada fazendo a seguinte fórmula (Andrey (2022)): célula da esquerda $\oplus$ (célula do meio $\vee$ célula da direita):

- 111 corresponde a 0, $1 \oplus (1 \vee 1) = 0$.
- 110 corresponde a 0, $1 \oplus (1 \vee 0) = 0$.
- 101 corresponde a 0, $1 \oplus (0 \vee 1) = 0$.
- 100 corresponde a 1, $1 \oplus (0 \vee 0) = 1$.
- 011 corresponde a 1, $0 \oplus (1 \vee 1) = 1$.
- 010 corresponde a 1, $0 \oplus (1 \vee 0) = 1$.
- 001 corresponde a 1, $0 \oplus (0 \vee 1) = 1$.
- 000 corresponde a 0, $0 \oplus (0 \vee 0) = 0$.

Assim, temos a seguinte ordem de *bits*, na base 2: $(00011110)_2$ e, para dar o nome desta regra, basta passar este para base 10: $(00011110)_2 = (30)_{10}$. Existe um total de 256 possibilidades (2^8), onde cada uma delas recebe o nome fazendo o mesmo processo que na 30. Na figura 2.2, encontra-se esta regra após realizados 15 passos através de uma célula (Weisstein (s.d.[b])).

Um dos fatores que torna esta regra tão interessante no contexto de aleatoriedade é o facto de ser caótica (Weisstein (s.d.[b])), ou seja, trata-se de uma regra altamente sensível às condições iniciais, uma vez que produz resultados diferentes com condições iniciais diferentes (Boccaletti et al. (2000)). A utilização desta regra para criar um **PRNG**, impõe que todo o processo seja determinista, no entanto, o caos presente nesta regra possibilita chegar a elevados níveis aleatoriedade, em alguns casos, com um caminho único (dependendo, novamente, das condições iniciais), pouca previsibilidade e ainda sem necessidade de um *input* aleatório, já que qualquer que seja o *input* o autómato celular segue um caminho único.

No entanto, os estudos que defendem que a regra é razoável para gerar números aleatórios são antigos. Estudos mais recentes defendem que a regra 30 é menos eficaz do que se pensava. Foi proposto por Hortensius um autómato celular não uniforme que era o conjunto da regra 90 e 150 e este apresentou uma melhor performance a gerar números aleatórios do que a regra 30, levando a que a qualidade desta regra seja referida como não suficiente (Tomassini e Perrenoud (2001)).

Na referência de Tomassini e Perrenoud (2001) é explicado que Wolfram propôs que os bits $k1, k2, ...$ a serem utilizados como chaves para cifrar a sequência de mensagens $p1, p2, ...$ são adquiridos através da sequência $(si(t), si(t+1), si(t+2), ...)$ obtida ao longo do tempo num sítio, neste caso i, da regra 30. Meier e Staffelback descobriram que existe uma forma de atacar este gerador em tempo razoável para tamanhos de chave menores ou igual a 500.

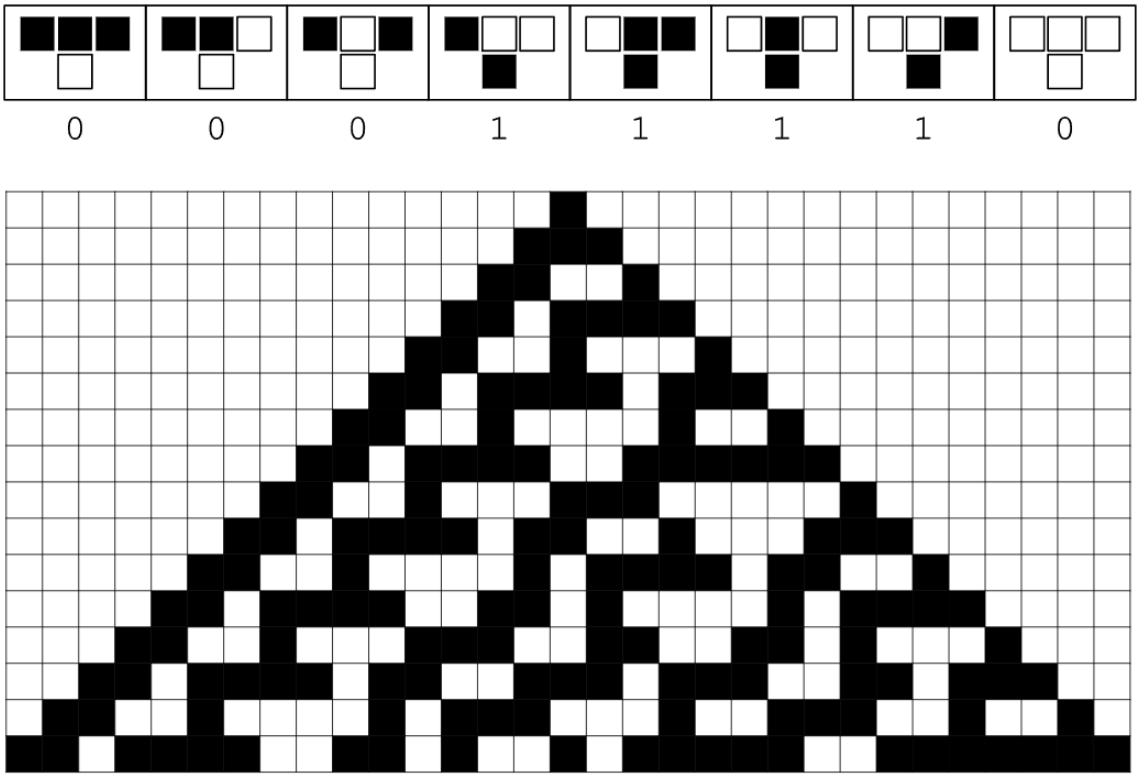
O ataque tem como objetivo reconstruir uma sequência de um dos lados direito ou esquerdo adjacente à sequência que se encontra num sítio especifico, adivinhando os estados dos sítios adjacentes. Ao conseguir aceder à sequência adjacente, torna-se fácil calcular de forma sucessiva os valores dos sítios que faltam decifrar (Tomassini e Perrenoud (2001)).

Como tal, apesar de antigamente a regra 30 ter sido considerada razoável para gerar números aleatórios, hoje em dia poderá não ser seguro utilizá-la para gerar estes números, devido à facilidade de ataque.

Outra regra relevante é a regra 110. Esta regra é considerada *turing complete*, ou seja, é considerado que consegue provar e computar tudo o que uma máquina *turing* consegue (a máquina *turing* é considerada como um modelo que consegue executar qualquer algoritmo) (Erlandsson (s.d.)).

Esta regra, tal como a 30, insere-se na classe 4 das regras de Wolfram, juntando a repetição da classe 2 e a aleatoriedade da classe 3 (e, consequentemente, da regra 30)

Figura 2.2: Regra 30 após 15 passos começando numa célula singular



(Shiffman (s.d.)). Assim, a regra 110 apresenta padrões, no entanto, estes padrões aparecem de forma aparentemente aleatória e imprevisível.

Na regra 110, foi estipulado o seguinte:

- 111 corresponde a 0.
- 110 corresponde a 1.
- 101 corresponde a 1.
- 100 corresponde a 0.
- 011 corresponde a 1.
- 010 corresponde a 1.
- 001 corresponde a 1.
- 000 corresponde a 0.

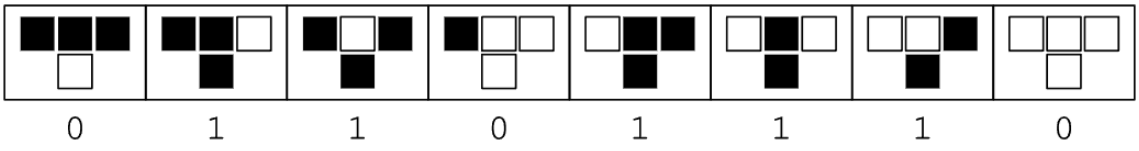
Aplica-se o mesmo procedimento da regra anterior: temos a seguinte ordem de *bits*, na base 2: $(01101110)_2$ que passando para base 10: $(001101110)_2 = (110)_{10}$. Na figura 2.3 (Weisstein (s.d.[a])) é possível ver o funcionamento da mesma.

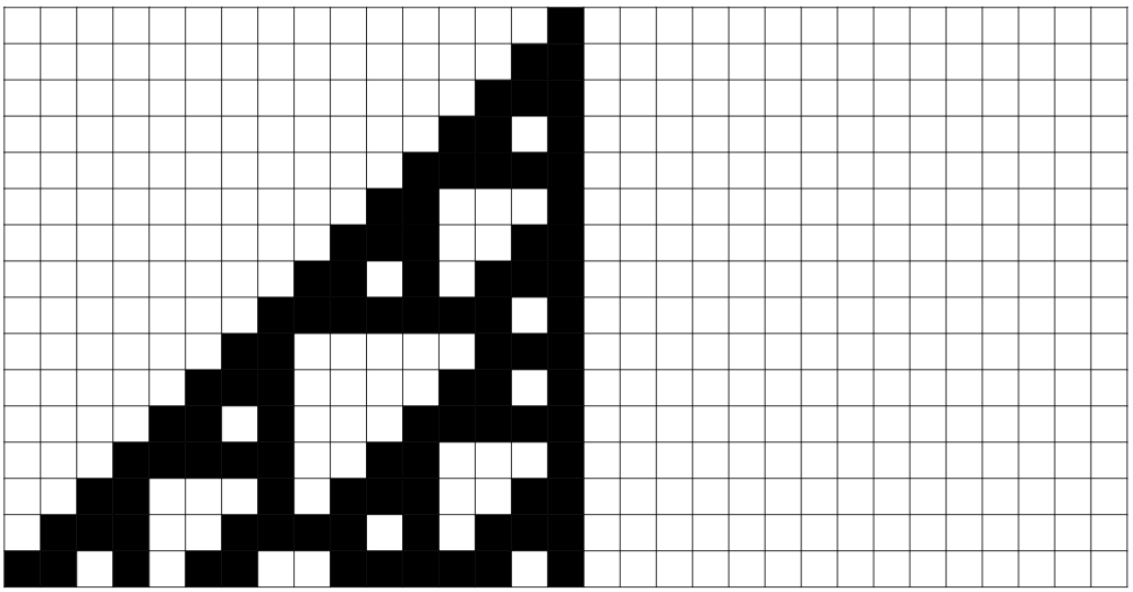
No livro escrito por Stephen Wolfram, o próprio defende que, após realizadas algumas experiências com a regra 110, apresenta-se um misto de regularidade com irregularidade (daí ser uma mistura da classe 2 com a 3) (Wolfram (2002)). Acrescenta ainda que durante o processo existiu um padrão de triângulos a aparecer de 7 em 7 passos e mais à frente de 80 em 80 passos tudo isto na parte esquerda de resultados. No lado direito, existem padrões mais irregulares apresentando, aparentemente, aleatoriedade durante as primeiras centenas de resultados.

No entanto, é percetível que, através da figura 2.4 , esta regra apresenta estruturas e padrões e, para além disso, a aleatoriedade desta regra depende fortemente das condições iniciais, o que indica que não é seguro para utilizar na criptografia. Depender das condições iniciais é algo inerente a qualquer regra, mas para ser segura, não deve existir uma dependência elevada das mesmas.

Tal como a regra 30, apesar de parecer ser uma boa hipótese para gerar aleatoriedade, não pode ser considerada segura para gerar números aleatórios. Como o objetivo deste estudo é criar autómatos celulares que provem ser aparentemente aleatórios, sendo dadas condições iniciais geradas de forma totalmente aleatória, e atendendo a que a regra 110 depende fortemente das condições iniciais e gera ainda demasiados padrões, a utilização desta regra não seria relevante.

Figura 2.3: Regra 110 após 15 passos começando numa célula singular





2.8.5 Autómatos Celulares e a sua Relação com Criptografia e Geração de Números Aleatórios

A utilização dos autómatos celulares para gerar números aleatórios não é novidade, uma vez que já foram utilizados em diversos estudos realizados sobre a origem de aleatoriedade de sistemas físicos (Wolfram (1986)). Um autómato tem um ponto de partida, ou uma condição inicial, e vai evoluindo seguindo as regras que lhe foram impostas.

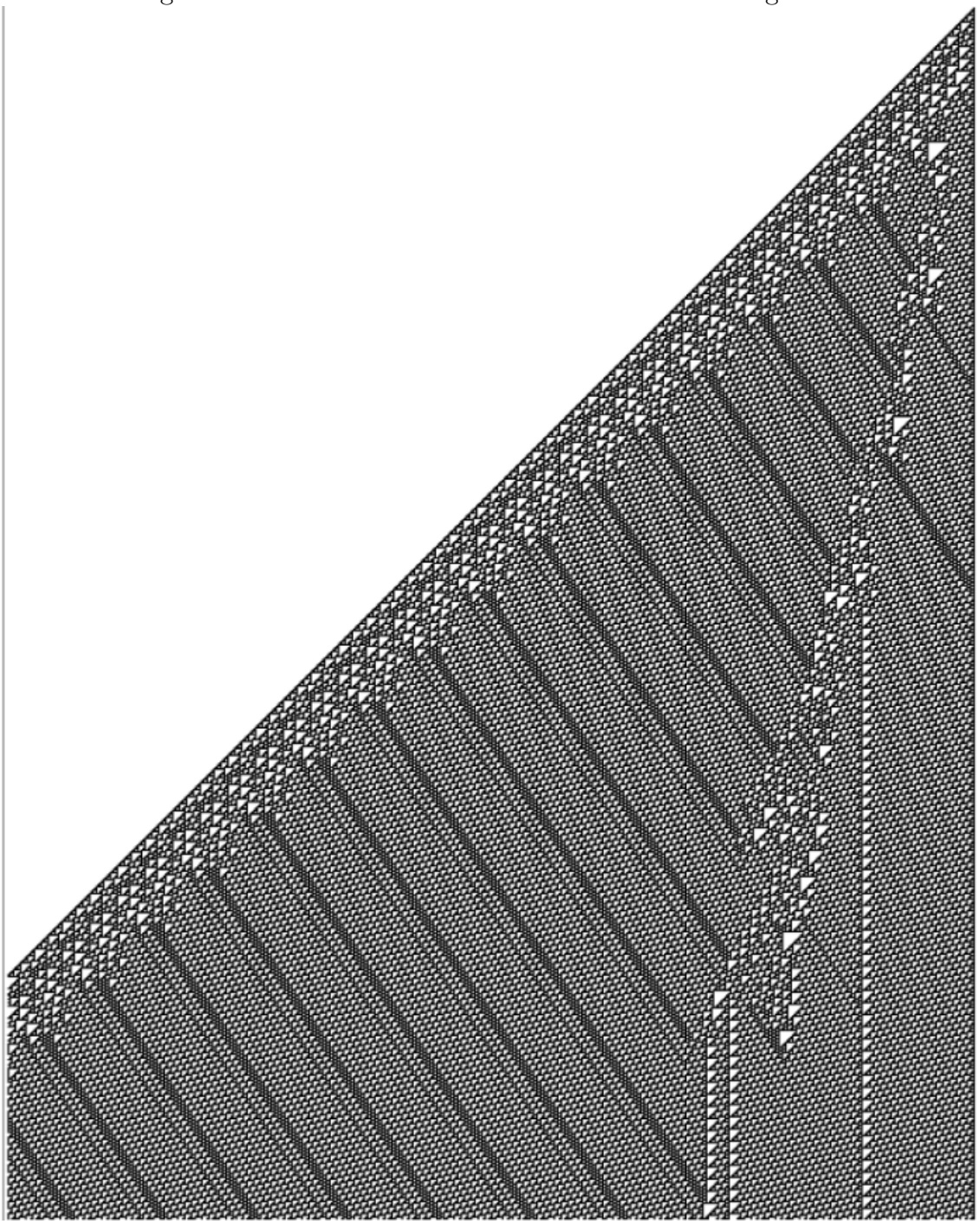
Assim, com diferentes regras e diferentes condições iniciais, tem-se diferentes resultados, podendo ter como *output* algo completamente previsível ou algo que parece ser aleatório (Wolfram (1986)). Posto isto, se o objetivo for utilizar autómatos celulares para a geração de números aleatórios é de interesse procurar resultados que parecem ser aleatórios. Será importante, portanto, escolher as regras adequadas e ter cuidado na escolha das condições iniciais do autómato. No entanto, se num autómato é preciso ter demasiado cuidado com a condição inicial que é dada, este poderá ser um indício de que a regra utilizada não é adequada para criar aleatoriedade, tal como na regra 110.

No entanto, no mesmo artigo referido no parágrafo anterior, é referido que mesmo quando as condições iniciais são demasiado simples, é possível ter resultados aparentemente aleatórios, o que leva a que a utilização de autómatos possa ser não só adequado mas sim recomendado em alguns casos. Para além dos resultados parecerem ser aleatórios, é necessário que a computação do autómato celular seja tão ou mais complexa e sofisticada que as previsões que serão realizadas no mesmo (Wolfram (1986)).

Outro grande ponto a favor da utilização dos autómatos celulares para os de classe 3 e alguns de classe 4 é o facto de serem computacionalmente irredutíveis, ou seja, não existe nenhum atalho para prever eficientemente o que o autómato irá gerar, a não ser que seja realizada a simulação passo a passo. Como tal, é difícil testar estes de forma eficaz e sem gastar demasiados recursos, já que pode exigir tempos de execução exponenciais, e poderá não funcionar, uma vez que alguns dos pontos a testar não têm resposta finita e não existe nenhum algoritmo que consiga responder (Wolfram (1986)). Um exemplo simples será o facto de poder ser impossível decidir se um certo estado é alcançável durante todo o processo do autómato.

Deste modo, compreendemos que os autómatos celulares, quando usados com as regras adequadas e, possivelmente, com condições iniciais favoráveis (mas não dependendo fortemente destas), podem ser apropriados para gerar números aleatórios.

Figura 2.4: Resultado de um teste realizado com a regra 110



2.9 Testes Estatísticos

Normalmente, o objetivo de um **TRNG** ou **PRNG**, num contexto de um sistema de segurança, é obter sequências aleatórias seguras de forma eficiente e eficaz. Portanto, este processo é, habitualmente, divido em dois. O primeiro trata-se de criar e escolher a natureza do gerador e o segundo passa por tentar ganhar confiança no mesmo, através de testes estatísticos. Existem outros tipos de testes, no entanto neste documento os testes de foco serão os estatísticos, uma vez que são mais comuns e os que normalmente ditam a confiança que se deposita num certo gerador.

Existem vários testes estatísticos disponíveis para avaliação dos geradores, que devem ser escolhidos cuidadosamente, considerando o contexto. Quantos mais testes forem concluídos com sucesso, maior será a confiança inerente na aleatoriedade do mesmo. Cada teste estatístico tem as suas próprias características, mas todos têm um objetivo em comum: analisar a sequência, ou um conjunto de sequências, fornecida pelo gerador em busca de comportamentos que demonstrem aleatoriedade ou a inexistência da mesma, como, por exemplo, padrões ou estruturas (Mattioli (2019)). Porém, é preciso ter em conta que mesmo que um **RNG** passe todos os testes a que foi submetido, nunca é possível dizer que é inquebrável, o que leva a que não seja ideal depositar demasiada confiança em qualquer gerador.

Os testes estatísticos são frequentemente utilizados para testar a aleatoriedade, sendo possível registar as sequências que tenham características aleatórias. Existem diversos conjuntos de testes que foram criados ao longo do tempo, sendo o primeiro relevante para este estudo o proposto por D. Knuth. Já o Instituto Nacional de Padrões e Tecnologia, ou *National Institute of Standards and Technology*/**NIST**, criou o padrão de testes SP 800-22 que contém 16 métodos de teste e um *Statistical Test Suite* do mesmo (Mengdi et al. (2021)) e existe ainda os testes *standard Die Hard* que tem interesse devido a se tratar de um conjunto de testes *standard* que avaliam a qualidade de um **RNG**. Tanto o primeiro como o segundo, contêm o teste mais importante para este estudo, o teste de frequências, que visa testar se uma certa sequência é independente e imprevisível e poderá também ser utilizada para perceber se segue uma distribuição uniforme.

Os testes estatísticos têm, neste âmbito, o propósito de demonstrar que um gerador é considerado de boa qualidade, dependendo da hipótese nula ou H_0 que, neste caso, será que o gerador a ser testado gera boas sequências aleatórias. Normalmente pega-se numa sequência relativamente grande de números, que foram gerados aleatoriamente, e testa-se H_0. Tendo em conta o que foi explicado na secção anterior, é fácil de perceber que H_0 é, no fundo, o estudo que determina se uma sequência gerada por **RNG** tem os aspetos necessários para ser considerado aleatória - os resultados devem ser independentes, deve respeitar a distribuição uniforme, devem ter valores imprevisíveis e não existência de correlação de dados.

No entanto, no presente estudo não serão utilizados testes de hipótese, mas será sim testado se um certo CA passa ou não a um critério mínimo. Este critério mínimo

será explicado ao longo deste documento.

2.10 Funcionamento dos Testes Estatísticos

Os testes estatísticos baseiam-se em testes de hipótese, onde a hipótese nula, H_0, é considerada uma tradição institucionalizada (Dar, Serlin e Omer (1994)). A ideia por detrás dos testes da hipótese nula, é assumir o argumento da hipótese como verdadeiro e comparar com os resultados obtidos do elemento que está a ser estudado. Se os valores dos resultados forem iguais ou inferiores ao valor de rejeição, normalmente chamado valor de significância e denotado como α, então a hipótese nula é rejeitada (Dar, Serlin e Omer (1994)), caso contrário é aceite.

Na documentação do **NIST**, a hipótese nula é definida por: H_0 : uma sequência é aleatória. Se esta for aceite quer dizer que a sequência é, de facto, aleatória. Caso contrário, a alternativa, H_1, será a aceite e a sequência será rejeitada e não considerada aleatória (Mattioli (2019)).

Este tipo de testes pode trazer alguns problemas se usados incorretamente. Existem dois tipos de problemas identificados nos testes de hipótese, o erro do tipo I que se trata da situação onde a hipótese nula deveria ter sido rejeitada e o elemento a ser estudado ser considerado bom e o erro do tipo II trata-se do contrário. No caso da aleatoriedade, o erro do tipo I indica que uma sequência aleatória acaba por ser considerada não aleatória e no erro do tipo II uma sequência não aleatória é considerada aleatória (Mattioli (2019)).

Estes problemas, ou erros, ocorrem devido à escolha indevida do valor de significância (Dar, Serlin e Omer (1994)), mostrando que o processo de escolha de um valor de significância deve ser realizado cuidadosamente. A escolha de um valor de significância indevido pode levar a que exista um dos tipos de erros explicados anteriormente e, como tal, poderá resultar em grandes consequências, principalmente se se tratar de um erro de tipo II (Mattioli (2019)). Isto, pois, se este erro ocorrer em informações criptografadas num banco, onde as informações de uma pessoa estão supostamente seguras através de um gerador que foi considerado aleatório erradamente, pode levar a que seja fácil para um adversário, ou atacante, intersetar e decifrar estas informações. Assim, o processo de escolha do valor de significância não deve, de todo, ser realizado de forma trivial.

Recapitulando, se tivermos em conta que foi escolhido um valor de significância, α, de forma devida, um gerador poderá ser considerado aleatório, através de um teste, quando o $p-value \geq \alpha$ aceitando a hipótese nula (Mattioli (2019)). Caso contrário, $p - value < \alpha$, a hipótese nula é rejeitada e a alternativa é aceite, resultando num gerador considerado não aleatório.

Um exemplo poderá ser: seja o argumento de uma hipótese nula que uma sequência é aleatória. Após realizados os testes para determinar um valor de significância adequado, seja este $\alpha = 0.05$, este indica que o esperado será existir 5 elementos em

100 que serão rejeitados. Se tivermos dois $p-values$, onde o primeiro é $p-value_1 = 0.06$ e o segundo $p - value_2 = 0.02$, significa que $p - value_1 \geq \alpha$ e $p - value_2 < \alpha$. Logo, o primeiro p_value não leva à rejeição da H_0 e seria considerada aleatória e, no segundo caso, rejeitava H_0 e a sequência que deu este $p - value_2$ é considerada não aleatória.

2.11 Tipos de Testes Estatísticos

Podemos agora explorar alguns dos testes disponíveis, nomeadamente alguns testes *standard* e a sequência de testes que o **NIST** propôs, seguindo a documentação deste. Serão apresentados o *Chi-squared test* de ajustamento e os propostos no **NIST**.

Os geradores devem produzir números com uma distribuição uniforme e, através de, por exemplo, o teste qui-quadrado, ou *Chi-squared test*, é possível chegar a uma conclusão (L'Ecuyer (1992)). O *Chi-squared Test* de ajustamento é um dos testes não paramétricos mais conhecidos e, como já foi referido anteriormente, tem como propósito assegurar que os números que são gerados estão distribuídos de forma uniforme. Caso o gerador passe a este teste, satisfará uma das propriedades de números aleatórios.

A sua fórmula é a seguinte: $X^2 = \frac{\sum(O_i - E_i)^2}{E_i}$, onde a soma é realizada sobre todas as categorias disponíveis. O elemento X^2 é o valor que se refere ao nome do teste, *Chi-squared*, o elemento O refere-se ao número real de entradas (frequência observada) na categoria, o elemento E refere-se ao número de entradas esperado (frequência esperada) (L'Ecuyer (1992)).

Uma forma de explicar os passos deste teste de forma simples poderá ser a seguinte (Afonso e Nunes (2019)):

1. Arranjar classes de valor, K, de X: $X : A_1, A_2, \ldots, A_n$.
2. Selecionar uma amostra aleatória e contar quantos elementos pertencem a cada classe A_i. Os valores obtidos são as frequências absolutas simples observadas, O_i.
3. Para cada classe A_i calcular a probabilidade teórica p_i^*, com base na distribuição definida em H_0, de conter elementos.
4. Em seguida, determinar as frequências absolutas estimadas E_i para cada classe A_i.
5. Se a distribuição teórica especificada em H_0 for adequada, então as frequências observadas aproximam-se das frequências estimadas.

Na referência de Turney (2022), existe um exemplo simples que mostra como funciona este teste, baseado na preferência de cães por certos sabores de comida.

Na tabela 2.1 podemos visualizar os dados referentes a este exemplo.

Sabor	Observado	Esperado
Garlic Blast	22	25
Blueberry Delight	30	25
Minty Munch	23	25

Tabela 2.1: Observações de sabores

Agora que temos os valores necessários podemos realizar os cálculos como um todo ou por partes. De forma a facilitar o processo os cálculos serão feitos por partes. O primeiro a ser realizado é o observado menos o esperado (O-E), que pode ser visualizado na tabela 2.2

Flavor	Observado	Esperado	O - E
Garlic Blast	22	25	-3
Blueberry Delight	30	25	5
Minty Munch	23	25	-2

Tabela 2.2: Cálculo O-E

De seguida podemos calcular o $(O-E)^2$, que se encontra na tabela 2.3.

Sabor	Observado	Esperado	$(O-E)^2$
Garlic Blast	22	25	9
Blueberry Delight	30	25	25
Minty Munch	23	25	4

Tabela 2.3: Cálculo $(O-E)^2$

Por fim, o último cálculo a ser realizado é o $(O-E)^2/E$ (que equivale ao *square*), que se encontra na tabela 2.4.

Agora podemos calcular o *square*, ou X^2 e tirar o resultado. Sabemos que que $X^2 = \frac{\sum(O_i-E_i)^2}{E_i}$, como tal basta somar todos os valores da tabela 2.4. Assim:

$X^2 = 0.36 + 1 + 0.16 = 1.52$

Neste momento, apesar de já termos o valor de X^2, não conseguimos tirar nenhuma conclusão. Como tal, agora temos de perceber se rejeitamos, ou não, a hipótese nula (Turney (2022)).

Para isso, em primeiro lugar precisamos de descobrir o valor de *square* crítico. Para calcular este valor temos de saber o valor de *df* (*degrees of freedom*) (números de grupos menos um) e o nível de significância (α) que, por convenção, é 0.05.

Assim, com $df = 2$ e $\alpha = 0.05$, através de uma tabela de distruibuição qui-quadrada percebemos que o valor crítico é aproximadamente 5.99 e, como tal, X^2 é menor do que o valor crítico.

Sabor	Observado	Esperado	$(O-E)^2/E$
Garlic Blast	22	25	0.36
Blueberry Delight	30	25	1
Minty Munch	23	25	0.16

Tabela 2.4: Cálculo $(O-E)^2/E$

Por fim, podemos passar à decisão de rejeitar, ou não, a hipótese nula. Para isso seguem-se os seguintes passos (Turney (2022)):

- Se o valor de *square* é maior que o valor crítico, então a diferença entre as distribuições observada e esperada é significativa, ou seja, $p < \alpha$. Assim, estamos perante um caso onde podemos rejeitar a hipótese nula;
- Caso contrário, a diferença entre as distribuições observada e esperada não é significativa, ou seja, $p > \alpha$. Assim, não podemos rejeitar a hipótese nula.

Para terminar o exemplo anterior, estamos perante um caso onde não podemos rejeitar a hipótese nula que diz que os cães escolhem os três sabores de forma uniforme, o que leva a concluir que os cães gostam dos três sabores.

Pegando ainda na conclusão interessante de Turney (2022), caso o dono da empresa que produz estes sabores estivesse a pensar em tirar, por exemplo, os sabores *Garlic Blast* e *Minty Munch*, porque pensava que os cães gostam menos, estaria errado.

Avançando agora para a sequência de testes proposta pelo **NIST**, temos os seguintes (Bassham III et al. (2010)):

1. Teste de Frequências - *Monobits*.
2. Teste de Frequências dentro de um Bloco.
3. Teste de Corridas.
4. Testes para a Maior Sequência de Uns num Bloco.
5. Teste da Matriz Binária de Postos.
6. Testes Espectrais.
7. Teste de Correspondência de Padrões Não Sobrepostos.
8. Teste de Correspondência de Padrões Sobrepostos.
9. Teste Estatístico Universal de Maurer.
10. Teste de Complexidade Linear.
11. Teste de Série.
12. Teste da Entropia Aproximada.
13. Teste de Somas Cumulativas (Cusums).

14. Teste de Excursões Aleatórias.
15. Teste Variante de Excursões Aleatórias.

Apesar de no presente estudo não serem utilizados estes testes, é importante ter conhecimento dos mesmos e do respetivo funcionamento. Na documentação de teste do **NIST** (Bassham III et al. (2010)) existe uma explicação detalhada de como utilizar estes testes, no entanto existem alguns fatores a ter em conta:

- Os testes de frequências devem ser os primeiros a ser realizados.
- Os testes que são apresentados em Bassham III et al. (2010) seguem uma distribuição padrão normal e uma distribuição de *Chi-Square*, $\chi^2 = \sum \frac{(o-e)^2}{e}$.
- A distribuição normal é utilizada para comparar o valor do teste estatístico, $p-value$, obtido através do **RNG** a ser testado, ao valor esperado da estatística que determina se é aleatório ou não.
- A distribuição qui-quadrada, X^2, é utilizada para comparar a qualidade do ajuste das frequências observadas numa sequência.
- Por fim, foram consideradas que as sequências utilizadas para testar são grandes, na ordem dos 10^3 até 10^7.

Só será apresentado o primeiro e o segundo teste, o dos *Monobits* e o teste de frequência dentro de um bloco, uma vez que são estes os mais importantes para o contexto do presente estudo. É proposto pelo **NIST** que o valor de significância será $\alpha = 0.01$.

2.11.1 Teste de Frequências *Monobits*

Uma das características de uma hipótese aleatória é o facto de que o número de zeros e de uns deve ser igual, ou praticamente igual. Segundo o **NIST**, o teste de frequências dos *monobits* tem como objetivo perceber se o número de zeros é igual ao número de uns e, caso seja, significa que seguem uma distribuição uniforme e passam ao teste. Tal como referido anteriormente, uma das características mais importantes de uma sequência aleatória é ter o mesmo número de zeros e uns, se uma sequência não passar a este teste, não fará sentido realizar mais nenhum (Bassham III et al. (2010)).

O algoritmo proposto pelo **NIST** para realizar este teste recebe dois *inputs*, o n que representa o tamanho da *string* de *bits* a ser testada, no caso dos *monobits* $n = 1$ e ϵ que representa a sequência a ser testada, onde $\forall i, \epsilon_i = \{0, 1\}$. Para além disso, a distribuição de referência é metade normal, já que se trata de um n grande.

Assim, existem condições para aplicar o algoritmo. Para tal, deve seguir-se os seguintes passos (Bassham III et al. (2010)):

1. Existirá agora um $S_n = X_1 + X_2 + \cdots + X_n$, onde cada um dos $X_i = 2\epsilon_i - 1$. Portanto, à partida, quanto mais perto o valor de S_n estiver, melhor será.

2. O valor estatístico, denotado por S_{obs} é o valor absoluto da soma que provém do valor da soma de X_i de toda a sequência a dividir pela raiz quadrada do tamanho da sequência. Ou seja, $s_{\text{obs}} = \frac{|S_n|}{\sqrt{n}}$.

3. De seguida calcula-se o $p - value = \text{erfc}\left(\frac{s_{\text{obs}}}{\sqrt{n}}\right)$, onde o $erfc$ representa o erro complementar.

Se $p - value < \alpha$, então a sequência é considerada não aleatória. Um exemplo dado na referência de Bassham III et al. (2010): Se tivermos uma sequência de *bits* $\epsilon = 1011010101$, com $n = 10$ então $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$, de seguida calculamos a $s_{\text{obs}} = \frac{|2|}{\sqrt{10}} = 0.63245532$. Por fim, calcula-se o $p - value = \text{erfc}\left(\frac{0.632455532}{\sqrt{2}}\right) = 0.527089$.

Neste caso, percebemos que $p - value \geq 0.01$. Como tal, a sequência é considerada aleatória.

Os testes de frequência em bloco têm como propósito testar a proporção ou probabilidade do aparecimento de um certo tamanho de blocos numa sequência. Enquanto que na anterior o número de 0s e 1s devia aproximar-se do valor $1/2$, neste caso deve aproximar-se do valor $M/2$, já que cada bloco é constituído por *M-bits* (Bassham III et al. (2010)). É recomendando que o número do n seja, no mínimo, de 100 *bits*, o $M \geq 20, M > 0.01n$ e $N < 100$.

Neste caso, o algoritmo recebe como *input* o n e o ϵ e ainda M, que se trata do tamanho do bloco a ser testado. Ao contrário do teste de frequências dos *monobits* este teste segue uma distribuição X^2. Os passos para realizar o algoritmo são os seguintes (Bassham III et al. (2010)):

1. Realizar uma partição da sequência a testar em $N = \lfloor \frac{n}{m} \rfloor$ blocos não sobrepostos e eliminar os *bits* não utilizados.

2. Determinar a proporção chamada de π_i dos elementos com o valor de 1 em cada bloco *M-bit*, utilizando a equação $\pi_i = \frac{\sum_{j=1}^{M} \epsilon(i-1)M+j}{M}$, para cada $1 \leq i \leq N$.

3. Calcular o valor estatístico de $X^2_{obs} = 4M\left(\sum_{i=1}^{N}\left(\pi_i - \frac{1}{2}\right)^2\right)$

4. Por fim, calcular o $p - value = igame\ (N/2, X^2_{obs}/2$. O valor *igame* é a função γ incompleta para $Q(a, x)$.

Dando novamente o exemplo que é dado neste artigo, se tivermos $n = 10, M = 3$ e $\epsilon = 0110011010$ então 3 blocos serão criados, o que significa que existiram $N = 3$ blocos com as seguintes sequências: $011, 001$ e 101, ignorando o último 0, calculamos o π_i de cada um: $\pi_1 = 2/3, \pi_2 = 1/3$ e $\pi_3 = 2/3$, depois calculamos $X^2_{obs} = 4 \times 3 \times ((2/3 - 1/2)^2 + (1/3 - 1/2)^2 + (2/3 - 1/2)^2) = 1$. Por fim, o $p - value = igame(3/2, 1/2) = 0.801252$.

Com estes valores, como o $p - value = 0.801252 \geq 0.01$, então a sequência é considerada aleatória.

A ideia que o **NIST** propôs para estes será utilizada, como referido, nos testes que serão feitos. No entanto, o algoritmo realizado é diferente, seguindo apenas a ideia de que o número de *M-bits* tem de se aproximar de $M/2$. Mais informações no capítulo de preparação de testes.

Capítulo 3

Trabalho desenvolvido

O objetivo será realizar um algoritmo generativo para criar autómatos celulares candidatos a ser um **PRNG**.

3.1 Definições e Conceitos

Esta secção tem como propósito dar uma introdução ao estudo que será feito de seguida, demonstrando alguns conceitos importantes e o contexto que dará suporte ao problema de investigação. Será seguida a referência de Ramos, Carapau e Correia (2022).

O *espaço de estados locais* refere-se a $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$, $n > 0$, a *aplicação local* (ou regra de um autómato celular, conceito da secção anterior) refere-se a $\phi : \mathbb{Z}_{\mathrm{m}}^{\mathrm{n}} \to \mathbb{Z}_n$, que determina as dinâmicas locais do sistema, e uma *configuração local* é qualquer elemento em $\mathbb{Z}_{\mathrm{m}}^{\mathrm{n}}$, ou seja, qualquer palavra ou bloco de tamanho m no alfabeto de $\mathbb{Z}_n$.

Seja $\phi_k : \mathbb{Z}_{\mathrm{n}}^{\mathrm{m+k}} \to \mathbb{Z}_{\mathrm{n}}^{\mathrm{k}}, k \in \mathbb{N}$, uma aplicação de blocos induzida pela *aplicação local* que transforma palavras em $\mathbb{Z}_n$, de tamanho $m+k$, em palavras de tamanho k, tendo em conta $\phi_k(\mathbf{x}_1, \ldots, \mathbf{x}_{m+k}) := \phi(\mathbf{x}_1, \ldots, \mathbf{x}_m)\phi(\mathbf{x}_2, \ldots, \mathbf{x}_{m+1}) \cdots \phi(\mathbf{x}_k, \ldots, \mathbf{x}_{m+k})$.

Considerando que m é um número ímpar, algo que fará bastante sentido principalmente para visualização e utilização de resultados do autómato, de forma a que $m = 2r + 1$, para um número qualquer natural r, a aplicação global é definida por:

$$\Phi : \mathbb{Z}_{\mathrm{n}}^{\mathbb{I}} \to \mathbb{Z}_{\mathrm{n}}^{\mathbb{I}}, \text{ onde } \Phi(x) := (\phi(x_{[j-r,j+r]})), \text{ com } j \in \mathbb{I}.$$

Na fórmula anterior, o $\mathbb{I}$ pode ser $\mathbb{Z}$, $\mathbb{N}$ ou um conjunto finito de $\mathbb{Z}$: $\mathbb{Z}_L = \{1, 2, \ldots, L\}$. O que será utilizado é o caso finito L.

Assim, podemos passar para a definição de autómato celular: trata-se da especificação do número de estados locais n, do tamanho da configuração local m, da aplicação local ϕ, do espaço de estados global $\mathbb{Z}^L$ e, eventualmente, as condições

de fronteira que dependem de $\mathbb{Z}^L$ e m (Ramos, Carapau e Correia (2022)). Deste modo, a evolução do sistema é dada pela iteração da aplicação global, Φ, dada uma condição inicial $x(0) = (x_i(0))_{i\in\mathbb{I}} \in \mathbb{Z}_{\mathrm{n}}^{\mathbb{I}}$:

$$x(t+1) = \Phi(x(t)), t \geq 0.$$

O parâmetro m introduz a dependência do próximo estado, numa certa posição, das r células à direita e à esquerda ($m = 2r + 1$). Em geral, é necessário especificar as condições de fronteira nos dois lados, sendo um exemplo caso o $\mathbb{I} = \mathbb{N}$ ou caso $\mathbb{I}$ seja um conjunto finito.

Seja $[j]_n$ a expansão em base n do número natural j, com o número de dígitos em $[j]_n$ fixo e igual a $l = n^m$. Assim, para $[j]_n = j_1 j_2 \dots j_l$ tem-se:

$$j = j_1 \times n^{l-1} + j_2 \times n^{l-2} + \ldots + j_{l-1} \times n^1 + j_l \times n^0.$$

Um número natural $j \in \mathbb{N}$ (onde $\langle j_1, \dots, j_l \rangle_n \in \mathbb{N}$ é representado por uma palavra $j_1, j_2, \dots, j_l \in \mathbb{Z}_{\mathrm{n}}^{\mathrm{l}}$ com $l \geq 1$. Assim, podemos caracterizar os lados direito e esquerdo do autómato celular:

$$j \in \mathbb{N} \to [j]_n = j_1, j_2, \dots, j_r \in \mathbb{Z}_{\mathrm{n}}^{\mathrm{r}},$$

$$j_1, j_2, \dots, j_r \in \mathbb{Z}_{\mathrm{n}}^{\mathrm{r}} \to \langle j_1, j_2, \dots, j_r \rangle_n = j \in \mathbb{N}$$

Após configurado o valor de m e ainda o espaço de configuração, $\mathbb{I}$, e os limites, tem-se um autómato celular totalmente caracterizado, com uma sequência correspondente à sequência de imagens de cada local de ϕ, ou da aplicação local, com o nome de *CA code rule*, ou regra do código do autómato celular (CA):

$$\alpha = (\alpha_1, \dots, \alpha_{n^m}) \in \mathbb{Z}_{\mathrm{n}}^{\mathrm{n^m}}$$

Esta sequência representa o autómato celular de forma funcional, que dá a um certo símbolo de uma certa posição que existe nesta sequência um significado funcional. Assim, e juntando os conceitos falados em cima, a posição j nesta sequência, α, apresenta a configuração onde a expansão n do número inteiro $(j-1)$ e o valor α_j é a imagem dessa configuração tendo em conta a regra da aplicação local ϕ:

$$\alpha = (\phi[j-1]_n)_{j=1}^{n^m}$$

Portanto, através desta sequência α é possível perceber qual é a regra do autómato celular. No entanto, é muito mais fácil e prático utilizar a *numeração de Wolfram* para o efeito, que já foi explicada na secção anterior (Ramos, Carapau e Correia (2022)).

3.2 Passos iniciais

O primeiro passo para a realização desta investigação é criar um algoritmo generativo principal, que é um programa feito através do computador que imita o processo evolucionário biológico de modo a resolver determinados problemas (Mitchell (1995)),

com o propósito de gerar possíveis candidatos a **PRNG**. Este algoritmo utiliza outros algoritmos nele incorporados. A execução do algoritmo generativo possibilita a recombinação de genótipos, que consistem em sequências de *strings* de *bits*, permitindo a geração de novos genótipos. Estes, após serem testados, são incorporados à população inicial, aumentando-a.

Neste sentido, existirá uma população inicial de *code rules* constituída de 15 elementos. Destes 15 elementos, apenas os que passarem ao critério mínimo poderão ser considerados para avançar para a nova população, chamada de população ideal final. É de notar que ao executar o algoritmo sempre com a mesma *code rule* poderá levar a diferentes resultados uma vez que o resultado é dependente também do ambiente em que foi realizado o mesmo. Como tal, a população ideal final conterá as melhores *code rules* mas tendo sempre em conta que foi preciso um certo ambiente para as mesmas serem incluídas. Estes conceitos serão estudados nas próximas subsecções.

O algoritmo genético será feito apenas com a recombinação e a mutação. Estes dois tipos de transformação podem ser feitos de forma relativamente diferente, tendo em conta o contexto, necessidades e resultados a obter. Serve a próxima subsecção para explicar estas duas transformações.

3.3 Noções utilizadas nas dinâmicas dos autómatos celulares

As populações contam com autómatos, dado um espaço de configuração, condição inicial e vizinhança. A cada iteração existe mudança na população, com a incorporação de novos exemplares, se forem considerados aptos. Cada autómato será encarado como um organismo, com um genótipo e um fenótipo (imagem resultante) que se trata das características típicas de realizações do referido autómato, envolvendo padrões, estruturas ou, idealmente no nosso caso, a falta destes (Carlos Ramos (2009)).

No presente documento, o algoritmo será executado para cada uma das *code rules* com 100 condições iniciais aleatórias diferentes, que geram fenótipos diferentes. A estes elementos podemos chamar de clones, uma vez que partilham o mesmo genótipo mas o ambiente é diferente. Se uma *code rule* em conjunto com um ambiente passar a certos testes então o fenótipo é guardado, apresentado visualmente e os genótipos considerados bons são guardados para recombinar.

Para o propósito do presente documento, os genótipos vão reproduzir-se entre si, ou seja, vão-se recombinar e, existirá também mutação de cada um dos genótipos. Comecemos pela mutação.

3.3.1 Mutação

A mutação é normalmente realizada alterando os valores de algumas posições de forma aleatória de uma *code rule*, ou dos genótipos (Mitchell (1995)). Enquanto que a recombinação tem o único propósito de reprodução entre dois organismos, a mutação poderá ter diversos propósitos, tendo em conta o contexto. Como no contexto do presente estudo, não faz sentido estudar *code rules* que não sigam uma distribuição uniforme, é realizada uma mutação nos genótipos de cada *code rule* de forma a que o número de zeros seja exatamente igual ao número de uns. Assim, esta condição necessária é assegurada à partida. Este processo poderia ser feito de forma diferente, criando uma população inicial de modo totalmente aleatória e acolher apenas aqueles que seguissem uma distribuição uniforme, no entanto as sequências terão 128 *bits*, 512 *bits* ou 32768 *bits*, como vamos ver de seguida, e a probabilidade de ter uma distribuição uniforme seria baixa.

De modo a criar uniformidade dos dados, é utilizado um algoritmo de mutação que recebe cada um destes elementos (*alpha_one*, *alpha_two*, ...), calcula o número de zeros e uns e determina qual destes valores está em minoria e, por fim, vai a uma posição aleatória de alguns elementos onde está o valor em maioria e troca-o pelo outro. A mutação irá modificar o mesmo α não criando um novo. Por exemplo, seja uma *code rule* com $m = 4$, portanto 16 valores (2^4), igual a:

$$\alpha = 0110000010000110$$

O algoritmo de mutação realiza o seguinte:

1. O algoritmo recebe uma *code rule*, neste caso α.
2. Realiza uma contagem de zeros e uns presentes na mesma. Para isso, a sequência é percorrida e a variável que guarda os zeros incrementa quando o valor atual é 0 e caso contrário a variável que guarda os uns incrementa.
3. O algoritmo verifica se o número de zeros e uns é igual. Se for retorna o mesmo α, sem nenhuma alteração.
4. Caso não exista uniformidade de dados, o valor que está em maioria na sequência é registado e apenas as posições que têm este valor serão modificadas para o outro valor. Se houver mais zeros o valor é 0, se não o valor é 1.
5. É criada uma lista que guarda as posições onde estão os valores em maioria.
6. É feito um *loop while* que corre até que o números de zeros e de uns seja exatamente o mesmo. Em cada iteração, uma posição aleatória da lista referida é escolhida e o valor é modificado para o valor que está em minoria e o número de zeros e uns é atualizado.
7. Quando o número de zeros é igual ao número de uns o α é retornado.

Pegando no exemplo anterior, como o número de zeros é igual a 11 e o número de uns é igual a 5, a lista vai guardar as posições onde os zeros estão e troca, a cada iteração,

para um valor igual a 1. Um exemplo da mutação de α seria $\alpha = 1110101010000110$, onde as posições 0, 4 e 6 foram modificadas de 0 para 1.

3.3.2 Recombinação

Depois de dois elementos terem sido mutados de modo a ambos seguirem uma distribuição uniforme, poderão ser submetidos à recombinação, imitando o processo biológico de reprodução. Um genótipo recombinado, vai ter características dos dois genótipos originais.

A recombinação consiste em selecionar dois elementos de um conjunto de *code rules*, aleatoriamente, e imitar a recombinação biológica entre dois organismos. Esta recombinação acontece com os elementos que avançaram da população inicial para a população ideal, no início, e, depois, existirá recombinação entre os novos elementos, com o objetivo de aumentar a população. Os elementos recombinados serão também submetidos aos mesmos testes que os elementos da população inicial e, se passarem, serão incluídos na população ideal.

A recombinação irá resultar num novo α que contém, tal como os originais, por exemplo para $m = 4$ novamente, 16 valores. O algoritmo de recombinação realiza o seguinte:

1. Recebe duas *code rules*.
2. É criado uma nova *code rule* que será devolvida.
3. De seguida, para cada um dos 16 *bits* da nova *code rule* será calculado um valor aleatório que pode ser 0 ou 1. Em cada iteração, se o valor for igual a 0, o *bit* recebe valor que se encontra no *bit* atual do α_1, caso contrário recebe o valor que se encontra no *bit* atual do α_2.

Por exemplo, se tivermos as seguintes *code rules* (note-se que já foi aplicada a mutação nos 2):

$$\alpha_1 = 0110011100101001$$

$$\alpha_2 = 1001100011010011$$

O algoritmo irá decidir, para cada um dos 16 valores do novo α_3, se vai buscar o *bit* correspondente ao primeiro parente ou ao segundo. Se a sequência que sai de escolha de valores aleatórios, que pode ser 0 ou 1 já explicado no ponto 3., for 0000011110101110, então:

$$\alpha_3 = 0110000010000011$$

No entanto, a recombinação não pode acabar por aqui. Como a recombinação acabou por fazer com que o número de zeros e uns esteja, novamente, desproporcional, já que existem 11 zeros, α_3 terá que ser mutado novamente. Como tal, existe um algoritmo que recebe dois α, utiliza o algoritmo descrito em cima e de seguida utiliza o algoritmo de mutação. No fim, retorna o novo α.

3.3.3 Conceitos

Esta subsecção tem como objetivo apresentar os conceitos fundamentais de um autómato celular de 2 estados, considerando os valores previamente definidos e os conceitos explicados na secção anterior.

Serão realizados testes com 3 valores de m diferentes, para que seja estudado o impacto da utilização de uma vizinhança maior, uma vez que, na teoria, leva a melhores resultados. Dado que, normalmente, se o m é maior então existe um maior grau de liberdade na escolha de autómatos com comportamentos distintos e variados. Em absoluto, quanto maior o m maior será o número de autómatos possíveis. Por um lado, poderão existir mais autómatos que não serão considerados relevantes uma vez que, como vamos perceber brevemente, apresentam padrões e estruturas, por outro lado existem mais possibilidades de escolhas de autómatos que quebram padrões existentes e assim mais candidatos a **PRNG**, com indicadores mais aperfeiçoados. Como tal, será realizado um teste em $m = 7$, $m = 9$ e $m = 15$, para perceber se realmente existe um efeito notável com estes 3 valores diferentes.

Para a população inicial de *code rules* que poderão ser candidatos a **PRNG**, foram criadas 15 *code rules* em $\{0, 1\}$ com $m = 7$, $m = 9$ e $m = 15$, ou seja, os testes serão realizados na expansão 2 ($n = 2$), ou com 2 estados. Foi escolhida a expansão 2 uma vez que torna simples e eficaz o processo de tornar a *code rule* uniforme, ou seja, ter o mesmo número de cada estado presente em cada *code rule*, neste caso mesmo número de zeros e uns, e os m foram os referidos pois, para além de serem números ímpares, geram sequências de comprimentos 128 *bits* para $m = 7$, 512 *bits* para $m = 9$ e 32768 *bits* para $m = 15$, com valores em $\{0, 1\}$.

Para $m = 7$:

1. Adaptando os conceitos da secção anterior, temos um espaço de estados locais $\mathbb{Z} = \{0, 1\}$, a aplicação local será $\phi : \mathbb{Z}_7^2 \to \mathbb{Z}_2$ e as configurações locais pertencem a $\mathbb{Z}_7^2$.

2. A aplicação de blocos induzida pela aplicação local é definida por $\phi_k : \mathbb{Z}_2^{7+\mathrm{k}} \to \mathbb{Z}_2^{\mathrm{k}}, k \in \mathbb{N}$, transforma palavras em $\mathbb{Z}_2$, de tamanho $7 + k$, em palavras de tamanho k, tendo em conta:

   $$\phi_k(\mathbf{x}_1, \ldots, \mathbf{x}_{7+k}) := \phi(\mathbf{x}_1, \ldots, \mathbf{x}_7)\phi(\mathbf{x}_2, \ldots, \mathbf{x}_8) \cdots \phi(\mathbf{x}_k, \ldots, \mathbf{x}_{k+7}).$$

3. Definindo $m = 2r + 1$, para qualquer número natural r, a aplicação global é definida por:

   $$\Phi : \mathbb{Z}_2^{\mathbb{I}} \to \mathbb{Z}_2^{\mathbb{I}}, \text{ onde } \Phi(x) := (\phi(x_{[j-r,j+r]})), \text{ com } j \in \mathbb{I}, \text{ onde } \mathbb{I} \text{ será } \mathbb{Z}_L = \{1, 2, \ldots, L\}.$$

4. A iteração da aplicação local, ϕ para dada uma condição inicial $x(0) = (x_i(0))_{i\in\mathbb{I}} \in \mathbb{Z}_2^{\mathbb{I}}$:

   $$x(t + 1) = \phi(x(t)), t \geq 0.$$

5. Recordando que $[j]_2$ representa a expansão em base 2 do número natural j, com o número de dígitos em $[j]_2$ fixo e igual a 2^7. Assim, o genótipo, ou *code rule*, associado à regra local será dado por

$$\alpha = (\phi[j-1]_2)_{j=1}^{2^7}$$

Para $m = 9$:

1. Espaço de estados locais $\mathbb{Z}_2 = \{0, 1\}$, a aplicação local será $\phi : \mathbb{Z}_2^9 \to \mathbb{Z}_2$ que, relembrando, determina as dinâmicas locais do sistema e, por fim, as configurações locais - palavras ou blocos de tamanho 9 no alfabeto de $\mathbb{Z}_2$), ou seja, elementos em $\mathbb{Z}_2^9$.

2. A aplicação de blocos induzido pela aplicação local é definida por $\phi_k : \mathbb{Z}_2^{9+\mathrm{k}} \to \mathbb{Z}_2^{\mathrm{k}}$, $k \in \mathbb{N}$ que tem como objetivo transformar palavras em $\mathbb{Z}_2$, de tamanho $9 + k$, em palavras de tamanho k, tendo em conta:

   $\phi_k(\mathbf{x}_1, \ldots, \mathbf{x}_{9+k}) := \phi(\mathbf{x}_1, \ldots, \mathbf{x}_9)\phi(\mathbf{x}_2, \ldots, \mathbf{x}_{10}) \cdots \phi(\mathbf{x}_9, \ldots, \mathbf{x}_{k+9})$.

3. Assim, definindo $m = 2r + 1$, para qualquer número natural r, a aplicação global é definida por:

$$\Phi : \mathbb{Z}_2^{\mathbb{I}} \to \mathbb{Z}_2^{\mathbb{I}}, \text{ onde } \Phi(x) := (\phi(x_{[j-r,j+r]})), \text{ com } j \in \mathbb{I}, \text{ onde } \mathbb{I} \text{ será } \mathbb{Z}_L = \{1, 2, \ldots, L\}.$$

4. De seguida, temos a iteração da aplicação global Φ que dita a evolução do sistema, para dada uma condição inicial $x(0) = (x_i(0))_{i\in\mathbb{I}} \in \mathbb{Z}_2^{\mathbb{I}}$:

$$x(t + 1) = \Phi(x(t)), t \geq 0.$$

5. Recordando que $[j]_2$ representa a expansão em base 2 do número natural j, com o número de dígitos em $[j]_2$ fixo e igual a 2^9. Assim, o genótipo, ou *code rule*, associado à regra local será dado por

$$\alpha = (\phi[j-1]_2)_{j=1}^{2^9}$$

Por fim, para $m = 15$:

1. Espaço de estados locais $\mathbb{Z} = \{0, 1\}$, a aplicação local será $\phi : \mathbb{Z}_{15}^2 \to \mathbb{Z}_2$ e as configurações locais em $\mathbb{Z}_{15}^2$.

2. A aplicação de blocos induzida pela aplicação local é definida por $\phi_k : \mathbb{Z}_2^{15+\mathrm{k}} \to \mathbb{Z}_2^{\mathrm{k}}, k \in \mathbb{N}$, transforma palavras em $\mathbb{Z}_2$, de tamanho $15 + k$, em palavras de tamanho k, tendo em conta:

   $\phi_k(\mathbf{x}_1, \ldots, \mathbf{x}_{15+k}) := \phi(\mathbf{x}_1, \ldots, \mathbf{x}_{15})\phi(\mathbf{x}_2, \ldots, \mathbf{x}_{16}) \cdots \phi(\mathbf{x}_1 5, \ldots, \mathbf{x}_{k+15})$.

3. Definindo $m = 2r + 1$, para qualquer número natural r, a aplicação global é definida por:

$$\Phi : \mathbb{Z}_2^{\mathbb{I}} \to \mathbb{Z}_2^{\mathbb{I}}, \text{ onde } \Phi(x) := (\phi(x_{[j-r,j+r]})), \text{ com } j \in \mathbb{I}, \text{ onde } \mathbb{I} \text{ será } \mathbb{Z}_L = \{1, 2, \ldots, L\}.$$

4. A iteração da aplicação global, Φ para dada uma condição inicial $x(0) = (x_i(0))_{i\in\mathbb{I}} \in \mathbb{Z}_2^{\mathbb{I}}$:

$$x(t+1) = \Phi(x(t)), t \geq 0.$$

5. Novamente, o genótipo, ou *code rule*, associado à regra local será dado por

$$\alpha = (\phi[j-1]_2)_{j=1}^{2^{15}}$$

Após todos estes passos temos um autómatos celulares totalmente caracterizados. Assim, as *code rules* são dadas por, respetivamente:

$$\alpha = (\alpha_1, \ldots, \alpha_{2^9}) \in \mathbb{Z}_2^{2^9}$$
$$\alpha = (\alpha_1, \ldots, \alpha_{2^7}) \in \mathbb{Z}_2^{2^7}$$
$$\alpha = (\alpha_1, \ldots, \alpha_{2^{15}}) \in \mathbb{Z}_2^{2^{15}}$$

No fundo, cada sequência α corresponde a diferentes configurações. Se convertermos o número $(j-1)$ para a sua representação em base 2, o valor α_j será o resultado da aplicação do autómato a essa configuração específica, de acordo com as definições da aplicação local ϕ.

3.3.4 Criação das *Code Rules*

Foram criados 15 elementos diferentes, com a designação genérica de *alpha* seguidos do seu índice, começando no *alpha_one* e terminando no *alpha_fifteen*. Como $n = 2$ e $m = 7$, $m = 9$ e $m = 15$ existem $2^7, 2^9, 2^{15}$ sequências possíveis, respetivamente. Cada um destes elementos terá 128, 512, 32768 *bits* cada um com o valor de 0 ou 1.

No começo, cada uma das *code rules* tem 128, 512 ou 32768 posições na forma de sequências, para cada um dos valores de m, e é dado, aleatoriamente, o valor de 0 ou 1 para cada uma destas, existindo uma probabilidade igual de sair o número 0 e o número 1 (por exemplo, $\{0010010 : 0\}$, no caso de $m = 7$, tendo os outros dois a mesma lógica). No entanto, uma das ideias básicas da aleatoriedade, e que é necessário para não só realizar os testes **NIST** como para ter os resultados desejados, é o facto de ser necessário que exista uniformidade dos dados (Mattioli (2019)). A uniformidade dos dados implica que o número de zeros seja igual ao número de uns, ou seja, cada um dos valores deve representar exatamente metade dos dados.

Não basta gerar aleatoriamente vários valores, zero ou um, para cada uma das sequências, uma vez que não é garantido, e é muito improvável, que exista um elemento que contenha uniformidade em todos os valores. Para que haja uniformidade de dados, será realizada a mutação referida anteriormente. De modo a criar uniformidade dos dados, é utilizado o algoritmo de mutação que recebe cada um destes elementos (*alpha_one*, *alpha_two*, ...) e, recapitulando, calcula o número de zeros e uns e determina qual destes valores está em minoria e, por fim, vai a uma posição aleatória de alguns elementos onde está o valor em maioria e troca-o pelo outro.

Quando existem 15 elementos, ou *code rules*, com uniformidade de valores, existem condições para correr o algoritmo principal. No entanto, primeiro será apresentado como é feita a decisão de entrada na população ideal final.

3.4 Testes e Entrada na População Ideal Final

É necessário escolher um critério para decidir se um certo autómato serve como candidato a **PRNG** ou não. Este tem de ser um critério simples e rápido de executar, tendo em conta que integrará o algoritmo genético.

O critério será o conjunto de dois testes. O primeiro é baseado no desvio de frequências de *strings* binárias, até tamanho 4, relativos aos valores teóricos da distribuição uniforme, explicada em cima. O segundo é o teste ao desvio padrão das frequências. Optou-se por escolher dois parâmetros de rejeição, δ, desvio de frequências, e δ_2, desvio padrão de frequências, que vão ditar se uma *code rule* entra ou não na população ideal final.

Em primeiro lugar, é necessário explicar como funciona o teste de frequências. Anteriormente, foi apresentado o procedimento proposto pelo **NIST**. O algoritmo aqui utilizado baseou-se na ideia apresentada pelo **NIST**, mas foi realizado de forma diferente, simplificada para ter maior rapidez de execução.

3.4.1 Aplicação dos Testes de Frequências

No teste de frequências implementado foi cumprida a exigência apresentada pelo **NIST**, que os *M-bits* tem de se aproximar de $M/2$.

Recapitulando, este teste tem duas vertentes - contagem dos *bits* únicos e contagem de blocos de *bits* (Mattioli (2019)). A primeira vertente, também chamada de teste de frequência *monobit*, consiste na contagem dos zeros e dos uns existentes na sequência que irá provir do algoritmo *ca_run*, que será explicado ao longo do documento. A percentagem de zeros e de uns deve ser o mais próxima da frequência esperada, que é $\frac{1}{2}$, tendo em conta que a aleatoriedade total é atingida quando a probabilidade de cada um deles é precisamente igual a este valor (Mattioli (2019)).

A segunda vertente consiste no mesmo teste para blocos, ou seja, em vez de ser feita a contagem de *bits* únicos, é feita a contagem de um conjunto de bits, por exemplo, se o objetivo for calcular a frequência dos blocos com tamanho 2, a contagem será feita aos blocos 00, 01, 10 e 11.

Seja f igual à frequência esperada, no caso do tamanho do bloco ser 2 existem 4 sequências possíveis pelo que a percentagem de cada um deles deve ser o mais próxima possível de $f = \frac{1}{4}$, para tamanho 3 tem de chegar o mais próximo possível de $f = \frac{1}{8}$ e, por fim, para tamanho 4 é necessário chegar o mais próximo possível de $f = \frac{1}{16}$. Para o problema de investigação, foi escolhido realizar estes testes até ao tamanho $k = 4$.

Como tal, existe um algoritmo que fará o cálculo destas frequências e retornar uma lista em Python, com 4 elementos:

1. Frequência dos *monobits* (0 e 1);
2. Frequência de cada uma das sequências possíveis para o caso do bloco com tamanho igual a 2 (00, 01, 10 e 11);
3. Frequência de cada uma das sequências possíveis para o caso do bloco com tamanho igual a 3 (000, 001, 010, 011, 100, 101, 110, 111);
4. Frequência de cada uma das sequências possíveis para o caso do bloco com tamanho igual a 4 (0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111).

Após existir uma lista com estes valores organizados, é criado outro algoritmo que irá agora calcular a proximidade da frequência de cada uma das sequências apresentadas em cima ao f correspondente, e retornar um vetor, ou uma lista, de 4 elementos:

1. No caso dos *monobits* os valores calculados são $|\frac{1}{2} -$ frequência de $0|, |\frac{1}{2} -$ frequência de $1|$
2. Para o bloco de tamanho 2 $|\frac{1}{4} -$ frequência de cada uma das sequências de tamanho $2|$.
3. Para o bloco de tamanho 3 $|\frac{1}{8} -$ frequência de cada uma das sequências de tamanho $3|$.
4. Para o bloco de tamanho 4 $|\frac{1}{16} -$ frequência de cada uma das sequências de tamanho $4|$.

É utilizado o módulo, de forma a que seja calculada a proximidade da frequência das sequências a cada um dos f. Como tal, quanto menor é o valor calculado, melhor é considerado o elemento.

3.4.2 Entrada na População Ideal Final

Em seguida explica-se o funcionamento geral do algoritmo *ca_run*. Dado um autómato a testar, considera-se um número, neste caso será 100, de listas diferentes através do algoritmo *ca_run*, uma vez que são dadas 100 condições diferentes aleatórias a cada autómato. Essas listas são o que possibilitam verificar o comportamento de um autómato celular tanto matematicamente como de forma visual, através de gráficos. Vamos chamar a cada uma destas listas de *temp*.

Dentro de uma *temp* existem outros elementos, que são também listas, uma vez que serão dados ao algoritmo 500 *steps*. Cada uma das 100 corridas tem um *temp* que é o valor resultante do algoritmo *ca_run*, constituído por 500 listas com 500 elementos cada, com valores em $\{0, 1\}$ que foram gerados aleatoriamente, onde o elemento 499

é estudado, para que o automáto celular esteja o mais longe possível da condição inicial. Se a posição 499 de cada um dos 500 elementos presentes na lista da *temp* passarem aos testes em todas as 100 iterações, então avança para os testes visuais. Caso contrário são automaticamente descartados.

Pegando, então, no elemento final de um *temp*, é construída uma lista que guarda vetores com os resultados dos testes explicados na subsecção anterior. Ou seja, o primeiro elemento conterá o resultado do teste da frequência dos *monobits*, dos uns e zeros e, como já sabemos, quanto mais próxima de 0 melhor, o segundo irá guardar o resultado teste de frequências para o bloco de tamanho 2, o terceiro guarda o resultado do teste de frequências para o bloco de tamanho 3 e, por fim, o quarto irá guardar o resultado do teste de frequências para o bloco de tamanho 4. A lista de vetores tem o seguinte formato: [[a, b], [a, b, c, d], [a, b, c, d, e, f, g, h], [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p]]. Já um exemplo da lista de vetores com valores reais pode ser o seguinte:

$$[[0.15400000000000003, 0.15400000000000003] \text{ no caso dos } \textit{monobits} \text{ e}$$

$$[0.187875751503006, 0.03456913827655311, 0.03456913827655311, 0.11873747494989978]$$
no caso do teste de frequências do bloco de tamanho 2.

Voltando ao processo da rejeição, ou não rejeição de um CA, cada vez que é criado uma destas listas de vetores será feito o cálculo da média de cada um dos elementos. No caso do primeiro vetor somamos o primeiro elemento com o segundo e dividimos por 2, no caso do bloco de tamanho 2 somamos o primeiro, o segundo, o terceiro e o quarto elemento e dividimos por 4, realizando este processo nos restantes dois. Existirá uma lista que irá guardar estas médias.

Será realizado ainda outro cálculo para testar se o autómato pode ser considerado um **PRNG**, uma vez que, em alguns testes que foram realizados apenas com este critério, entravam resultados irrelevantes, havendo necessidade de criar um maior rigor relativo ao critério de admissão na população ideal final. Este será o cálculo do maior desvio padrão existente nas 100 *temps* originadas por cada autómato.

Como tal, tendo uma sequência binária ϵ se os valores dos testes realizados forem menores que o valor de rejeição, ou valor de significância, δ e δ_2, respetivamente, então o CA é considerado como um candidato a **PRNG**, caso contrário é descartado. Note-se que estes critérios podem permitir eliminar bons autómatos e até incluir autómatos menos bons, no entanto, neste momento o objetivo é gerar uma população de candidatos relevantes a **PRNG**, sendo posteriormente submetidos a testes finais **NIST**, num trabalho futuro.

Como referido anteriormente, o valor de significância deve ser escolhido com alguma cautela. O primeiro foi estimado, realizando o teste das frequências novamente, mas com valores da função aleatória *standard* do Python chamada *numpy*, e o módulo mais seguro para **PRNGs** chamado *secrets*. Com estas duas vertentes, foram feitas duas listas, correspondente a cada um deles, com 100 elementos com 500 valores

cada um em $\{0, 1\}$, de modo a imitar o processo do problema de investigação. Já para o valor de δ_2 foi realizado um teste semelhante, mas estudando o desvio padrão.

No caso da biblioteca *numpy*, obteve-se um valor de desvio aproximado de 0.047 e um desvio padrão aproximado de 0.038. Portanto, é possível considerar $\delta = 0.05$ e $\delta_2 = 0.04$ respetivamente. No que diz respeito à biblioteca *secrets*, o valor de desvio aproximado foi de 0.049 e o desvio padrão aproximado foi de 0.039. De forma semelhante ao caso anterior, podemos considerar $\delta = 0.05$ e $\delta_2 = 0.04$. Assim, para efeitos de parâmetros de rejeição, serão adotados os valores de $\delta = 0.05$ e $\delta_2 = 0.04$.

É importante observar que, uma vez que estamos a realizar uma pré-seleção, podemos adotar valores de δ e δ_2 mais aproximados.

3.5 Funcionamento do Algoritmo e Expectativa de Resultados

Serve esta secção para fornecer uma explicação detalhada sobre o funcionamento do algoritmo e os passos essenciais para alcançar o resultado desejado. Este resultado consiste numa população designada como "população ideal final", a qual abrange as *code rules* que se apresentam como candidatas a **PRNG**.

3.5.1 Entrada de Elementos da População Inicial na População Ideal Final

O algoritmo principal, *run_code_rules_with_time* recebe uma população (*population*), o número de condições iniciais aleatórias arbitrário (*len_randoms*) e o tempo (t) também arbitrário que o programa terá para correr.

Este algoritmo receberá sempre a população inicial, denominada como *population0*, 100 condições aleatórias iniciais (*len_randoms*) e 1000 segundos (t). É de notar que quanto maior for o tempo de execução, maior será a população ideal final, mas para fins de teste 1000 segundos são suficientes. O programa irá executar enquanto o tempo desde o começo até uma certa altura seja menor que t. Até que o tempo termine, será invocado o algoritmo (*run_and_test_every_code_rule*), que contém toda a lógica para testar cada população ou elemento recombinado que é dado, que será da população inicial ou apenas uma *code rule* que provém da recombinação entre dois organismos (entre outras duas *code rules*), respetivamente.

O algoritmo invocado receberá a população a testar (*population*), o número de condições iniciais (*len_randoms*), uma lista que irá guardar todos os vetores de frequências (*vector_collection_final*), um dicionário que guarda como chaves as *code rules* e como valor os vetores (*dic_vector_collection_final*), uma variável para guardar quantas vezes o programa já correu (*indexes_of_runs*) e, por fim, uma variável que pode ter o valor 0 ou 1, sendo 0 quando se trata de testar uma população inicial e 1 quando se trata de testar uma recombinação entre dois genótipos.

O algoritmo que contém a lógica para criar um autómato celular tem o nome *ca_run* e recebe uma lista de números aleatórios (*lis*) (ou seja, a condição inicial), que:

- Contém 100 elementos, listas que contêm, cada uma, 500 valores que podem ser 0 ou 1.
- Destes 100 elementos é escolhido um índice aleatório (é gerado de forma aleatória um índice entre 0 e 99) e a lista que está nesse índice é utilizada como condição inicial.

Recebe o número de passos e devolve o mesmo número de listas que o número de passos fornecido. Ou seja, como serão dados 500 passos no algoritmo, este irá criar 500 listas diferentes. Como já referido, cada conjunto destes de 500 listas tem o nome de *temp*. O conjunto de elementos do *temp* é considerado um CA e serão estes os valores utilizados para gerar um gráfico.

Recebe também os limites de vizinhança à esquerda e à direita, que serão gerados de forma aleatória, com valores entre 0 e 1. Será do tipo $[x, y, z]$ para $m = 7$, $[x, y, z, w]$ para $m = 9$ e $[x, y, z, w, v, t, o]$ para $m = 15$. Serão iguais tanto à esquerda como à direita, onde x, y e z receberão de forma aleatória entre um valor entre 0 e 10, inclusive. Esta escolha foi arbitrária, uma vez que não fará muita diferença nos resultados.

Recebe também os limites fronteira. De modo a explicar, tendo $x_i(t)$ como estado da célula i no instante t, temos:

$$x(0) = x1(0)x2(0)x3(0)$$

No caso de $m = 3$, temos:

$$x(1) = x1(1)x2(1)x3(1)$$

Seja ϕ a aplicação local que define a regra. O estado seguinte é sempre determinado por:

$$x1(1) = \phi(xL, x1(0), x2(0))$$
$$x2(1) = \phi(x1(0), x2(0), x3(0))$$
$$x3(1) = \phi(x2(0), x3(0), xR)$$

Como tal, é necessário especificar xL e xR. No caso $m = 3$ são as condições fronteira à esquerda L e direita R. No caso de $m = 5$, a aplicação local, ϕ pega em 5 estados vizinhos e determina 1 estado seguinte:

$$x1(1) = \phi(xL1, xL2, x1(0), x2(0), x3(0))$$
$$x2(1) = \phi(xL2, x1(0), x2(0), x3(0), xR1)$$
$$x3(1) = \phi(x1(0), x2(0), x3(0), xR1, xR2)$$

Concluindo, é necessário especificar $xL1, xL2$ à esquerda (2 valores = $(m-1)/2$) e $xR1$, $xR2$ à direita.

Como tal, para $m = 7$ é necessário indicar 3 valores à esquerda e à direita, para $m = 9$ é necessário indicar 4 valores e para $m = 15$ é necessário indicar 7. Assim, é criada uma lista com 3 valores, 4 ou 7, respetivamente, e cada um dos elementos receberá os únicos valores possíveis, 0 ou 1.

Por fim, cada uma destas code rules terá 100 *temps* e todos os elementos finais de cada um destes resultados será testado com o teste de frequências e será o que poderá dar entrada na população ideal final, caso passem ao critério mínimo (junção dos dois testes). Caso contrário, tanto a *code rule* como os resultados desta (*temp*) serão imediatamente descartados.

Durante cada uma destas iterações, são gerados os vetores de frequência para o elemento 499 de cada um dos 100 *temp*. Estes vetores são registados para posterior utilização nos testes especificados anteriormente. É utilizada uma estrutura de dados para armazenar a análise dos vetores de frequência para cada *code rule*, na qual estão 100 sub-listas. Cada sub-lista contém o resultado da análise dos vetores de frequência, proveniente da execução do teste de frequência. Para simplificar a explicação, seja a lista que contém as 100 sub-listas chamada **lista final de vetores**.

Agora, existem todas as condições necessárias para começar a realizar os testes e perceber quais serão as *code rules* da população inicial que serão aceites na nova população (população ideal final). Será então calculada a média para cada um dos índices dos vetores das 100 sub-listas que originam da lista de vetores final de vetores.

De modo a terminar este teste, basta verificar se $a < \delta, b < \delta, c < \delta$ e $d < \delta$, com $\delta = 0.05$, onde a, b, c e o d representam os valores da lista que contém as frequências explicadas anteriormente ($[a, b, c, d]$). Basta um destes não ser menor que δ para que o CA seja descartado. De seguida, o desvio padrão de cada um será testado e será verificado se é menor que $\delta_2 = 0.04$. Como tal, é calculado o desvio padrão para cada uma das 100 sub-listas e é retirado o desvio padrão maior e, caso este seja menor que δ_2, então a *code rule* passa a este teste.

Uma *code rule* é adicionada à população ideal final se e somente se passar nos dois testes. No entanto, se apenas uma ou menos *code rules* forem adicionadas à população ideal, é criada uma nova população inicial, a população ideal final fica vazia e é tudo gerado novamente e o tempo começa a contar novamente, uma vez que com zero ou uma *code rule* na população ideal nunca seria possível existir recombinação e continuar a evoluir a população.

Entrada de Elementos Recombinados na População Ideal final

Quando é certo que existe mais do que uma *code rule* na população ideal, inicia-se uma nova fase do algoritmo, que se trata da recombinação entre elementos da população e são realizados, novamente, os mesmos testes nestes. A recombinação

é realizada escolhendo, de forma aleatória, dois elementos da população ideal a cada iteração, aplicar o algoritmo de recombinação e de mutação de uniformidade de dados, referenciados anteriormente, sobre a nova *code rule* e correr o algoritmo *run_and_test_every_code_rule* com a nova *code rule*.

Sempre que dois elementos são recombinados entre si, não poderão recombinar novamente uma com a outra. Assim, é evitado que existam α, ou *code rules*, iguais ou demasiado parecidas, já que os genótipos pais seriam os mesmos e existiria comportamentos muito parecidos devido à existência de "irmãos". A entrada de elementos recombinados na população ideal assemelha-se à entrada dos elementos da população inicial, tendo de passar, também, pelos mesmos dois testes e necessitando de apresentar valores menores que os *deltas* explicados anteriormente.

3.6 Preparação de testes e Demonstração Gráfica

Por fim, o objetivo será realizar todo o processo descrito e criar gráficos para visualizar e verificar se os autómato celulares criados através das *code rules* da população ideal em conjunto com uma certa condição inicial, geraram bons candidatos a **PRNG**. O que irá ditar se estes conjuntos são de interesse para esta investigação é a sua aparência, precisando de ser aparentemente aleatória sem qualquer tipo de estrutura ou padrão.

A criação de gráficos é realizada através da biblioteca matplotlib.pyplot, do Python, que permite que sejam criadas várias formas de visualizações. A utilização desta biblioteca permite apresentar e estudar os resultados de forma visual e facilita a representação visual de dados. Para que sejam apresentados os gráficos será dada uma das 100 *temps* que cada uma das *code rules* obteve. Para cada uma, basta visualizar um dos resultados, como tal é escolhido um índice aleatório da lista de *temps*, entre 0 e 99, e será apresentado este.

A ausência de estruturas dita a aleatoriedade, uma vez que estas têm impacto nas distribuições das palavras de tamanho k que ocorrem e correlações entre as sucessões de palavras que ocorrem ao longo do tempo. Deste modo, a presença de estrutura significa, normalmente, um afastamento da distribuição uniforme para uma das distribuições de $k = \{1, 2, 3, 4\}$ (como já referido anteriormente, o teste de frequência de *monobits* e teste de frequências de um bloco).

No entanto, pode acontecer que algum autómato celular com estrutura visualmente identificada possa ser considerado nos testes realizados. Para retirar estes autómatos seria necessário utilizar os testes **NIST**. O presente estudo foca-se em produzir um algoritmo generativo que dá origem, de forma simplificada, a bons candidatos a **PRNG**.

Para além disso, existe ainda um sistema de pontos, onde cada uma das *code rules* recebe uma pontuação consoante a média das frequências, isto pois quanto menor é o valor calculado do teste de frequências dos *monobits* e dos teste de blocos, melhor

é considerado o elemento já que se aproxima da distribuição uniforme. Como tal, o último algoritmo que é utilizado no programa recebe todos os vetores de média que uma *code rule* obteve e realiza a soma de todos os elementos. A metade de *code rules* que apresentar um valor menor é considerada melhor e a a segunda metade é considerada a pior. No entanto, algumas vezes, apesar da metade considerada pior ter uma performance pior em termos de pontuação, esta obtém melhores resultados gráficos aparentes do que a outra metade. Contudo, os elementos da pior metade estão sempre mais afastados da distribuição uniforme.

Cada uma das execuções vai gerar x autómatos diferentes onde a metade melhor tem os autómatos que geraram um valor de médias total mais baixo e a pior metade com os restantes. Consideram-se as melhores metades como melhores valores. Todavia, caso existam casos da pior metade que apresentem uma aparência aparentemente aleatória, serão também incluídos uma vez que, matematicamente, os valores não ficam demasiado distantes uns dos outros. No decorrer de cada execução deste programa, os resultados são armazenados e apresentados.

Capítulo 4

Resultados

Neste capítulo, serão apresentadas figuras que demonstram os resultados obtidos com os algoritmos desenvolvidos, tabelas que apresentam os valores dos testes e será realizada uma discussão abrangente sobre os resultados, explorando as suas implicações e relevância para a geração de números aleatórios. Devido ao facto de existirem diversos resultados similares, não serão todos apresentados, no entanto serão apresentados tanto gráficos que apresentam bons resultados como outros que apresentaram maus resultados.

Como referido anteriormente, existe uma população inicial com 15 *code rules*, uniformes, geradas aleatoriamente, candidatas a **PRNG**. O algoritmo principal irá correr utilizando esta população e serão dadas 100 condições iniciais aleatórias e, por fim, 1000 segundos para que a população final, ou ideal, tenha um número razoável de elementos. Será relevante mencionar que sempre que o programa é executado, é gerada uma nova população inicial e a população ideal é também recriada.

Recapitulando, apenas as *code rules* que passarem aos dois testes de frequências descritos anteriormente farão parte da população ideal. Por fim, tanto os resultados que provêm da população inicial como os do resultado da recombinação entre elementos serão apresentados.

Serão realizadas 5 execuções, para cada um dos m já referidos, de modo a perceber se um maior m implica melhores resultados, que será discutido na secção seguinte, na Discussão de Resultados. Em cada uma destas, serão apresentados os resultados das mesmas. Novamente, cada uma das *code rules* que foram consideradas aceites, passaram aos testes, e que portanto estão na população ideal final, têm 100 resultados diferentes, *temps*, possíveis de mostrar. Como tal, será escolhido, de forma aleatória, um número entre 0 e $len\_randoms - 1$, neste caso igual a 99, e será apresentado o resultado visual que corresponde a esse índice para todos as *code rules*. O código implementado em Python será colocado como anexo.

4.1 Resultados para $m = 7$

Serve esta subsecção para apresentar os resultados das 5 execuções realizadas para $m = 7$.

4.1.1 Primeira Execução

Na primeira execução, a população de candidatos contou com 8 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7\}$ onde os dois primeiros elementos vieram da população inicial, e as recombinações foram as seguintes: dos índices [1, 0], [2, 0], [3, 2], [0, 3], [2, 4] e [4, 5], representando os últimos 6 elementos.

Na tabela 4.1, podemos visualizar a média que cada um dos índices da população ideal obteve e o respetivo resultado, aproximados.

Segundo esta pontuação, as *code rules* de forma ordenada são: $\{P_{candidatos}[6], P_{candidatos}[5], P_{candidatos}[1], P_{candidatos}[7], P_{candidatos}[4], P_{candidatos}[0], P_{candidatos}[2], P_{candidatos}[3]\}$ sendo que a melhor metade contém os índices $\{6, 5, 1, 7\}$ e a pior os restantes, $\{4, 0, 2, 3\}$. A tabela 4.2 tem os valores dos desvios padrão e a tabela 4.3 apresenta as condições fronteira

Na apresentação desta execução as *code rules* serão apresentadas em base 32 uma vez que se tratam de 128 sequências. As *code rules* são:

1. $P_{candidatos}[0] = 55LB7A39U47VTKN6LG3M8A4KMH$
2. $P_{candidatos}[1] = 4MCIU2SSVA1SJQEB7S6OHKQO46$
3. $P_{candidatos}[2] = 55KBV21OV81THS77NO2MOCSS4L$
4. $P_{candidatos}[3] = 55LBFA39U83VHSN6LO2M8C4KML$
5. $P_{candidatos}[4] = 55KBVA18V83VHON7NO2M84KS6L$
6. $P_{candidatos}[5] = 55L17A39U87VTSN6LG2M8E4KML$
7. $P_{candidatos}[6] = 55KBVA18V83THS77NO2MO4SS4L$
8. $P_{candidatos}[7] = 55L1FA18U87VLSN6LG6M8EKS6L$

Note-se que no início, a recombinação é apenas realizada com os valores disponíveis que vieram da população inicial, no caso anterior os índices $\{0, 1\}$. À medida que novos elementos são adicionados, existem mais possibilidades de recombinar, sendo totalmente possível que exista recombinação entre elementos que foram originados de uma recombinação, sem nunca haver repetição de índices recombinados.

Em termos de gráficos, todos foram considerados aparentemente aleatórios, uma vez que não apresentam nenhuma estrutura ou padrão, algo que não é comum e que permitiu ter um grande número de resultados relevantes. Em todos podemos visualizar que não existe qualquer tipo de padrão ou estrutura e serão todos considerados

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.019	0.015	0.012	0.009	0.054
$P_{candidatos}[1]$	0.016	0.013	0.011	0.008	0.050
$P_{candidatos}[2]$	0.022	0.016	0.011	0.006	0.056
$P_{candidatos}[3]$	0.030	0.018	0.016	0.010	0.074
$P_{candidatos}[4]$	0.018	0.015	0.012	0.007	0.051
$P_{candidatos}[5]$	0.014	0.008	0.008	0.008	0.037
$P_{candidatos}[6]$	0.010	0.008	0.009	0.006	0.032
$P_{candidatos}[7]$	0.016	0.016	0.009	0.008	0.049

Tabela 4.1: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidatos}[x]$	Desvios Padrão
$P_{candidatos}[0]$	0.020
$P_{candidatos}[1]$	0.025
$P_{candidatos}[2]$	0.011
$P_{candidatos}[3]$	0.009
$P_{candidatos}[4]$	0.007
$P_{candidatos}[5]$	0.006
$P_{candidatos}[6]$	0.004
$P_{candidatos}[7]$	0.008

Tabela 4.2: Desvios padrão aproximados para cada $P_{candidato}[x]$

como bons e inseridos diretamente na população ideal final. Como exemplo, temos o elemento 0 e 1 nas figuras 4.1 e 4.2, respetivamente.

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 0, 0]
$P_{\text{candidato}}[1]$	[0, 0, 0]
$P_{\text{candidato}}[2]$	[0, 0, 0]
$P_{\text{candidato}}[3]$	[0, 0, 0]
$P_{\text{candidato}}[4]$	[0, 0, 0]
$P_{\text{candidato}}[5]$	[0, 0, 0]
$P_{\text{candidato}}[6]$	[0, 0, 0]
$P_{\text{candidato}}[7]$	[0, 0, 0]

Tabela 4.3: Condições de Fronteira cada $P_{candidato}[x]$

Figura 4.1: Gráfico da Primeira *Code Rule*, $P_{candidato}[0]$, da Primeira Execução para $m = 7$

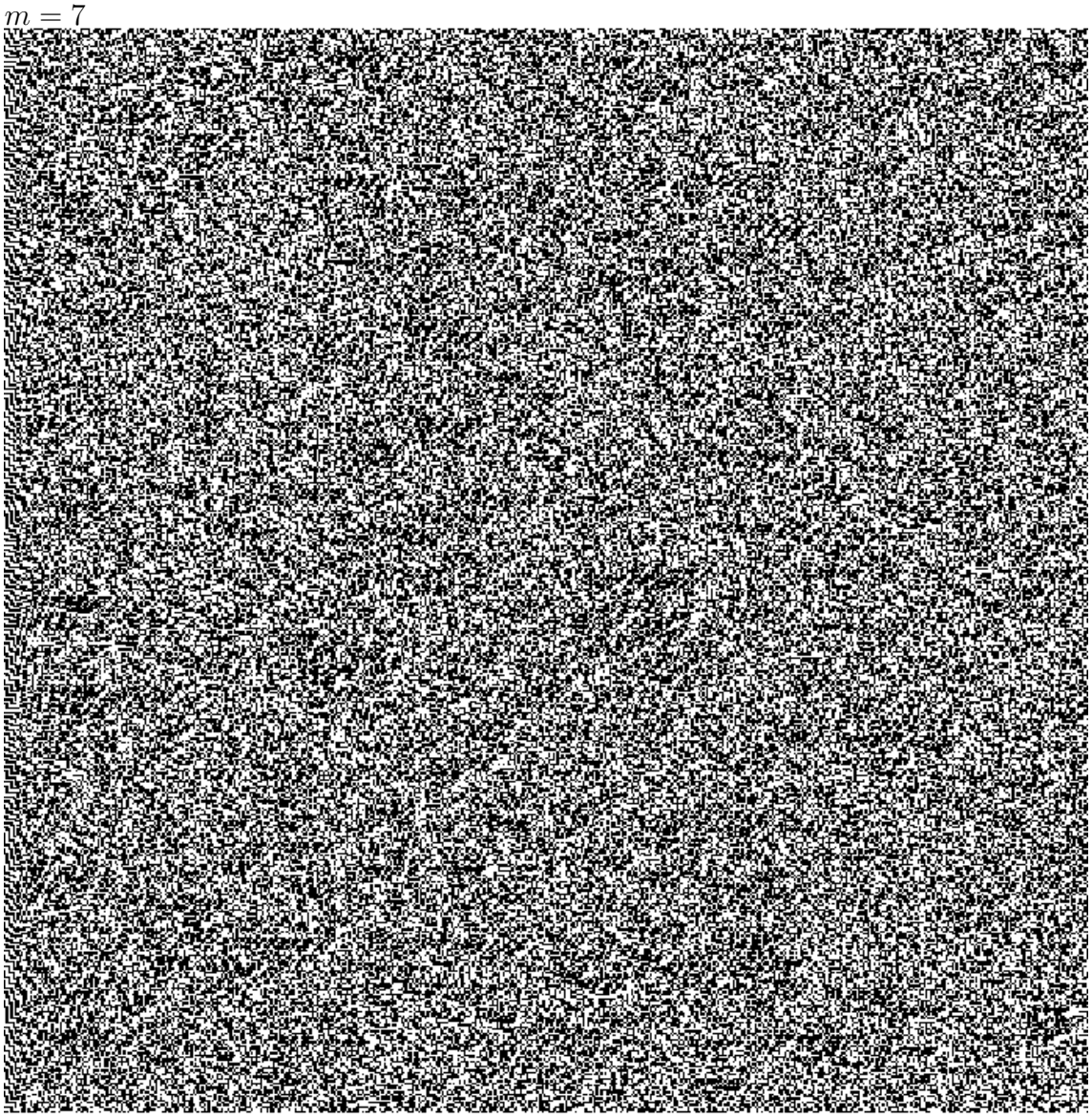
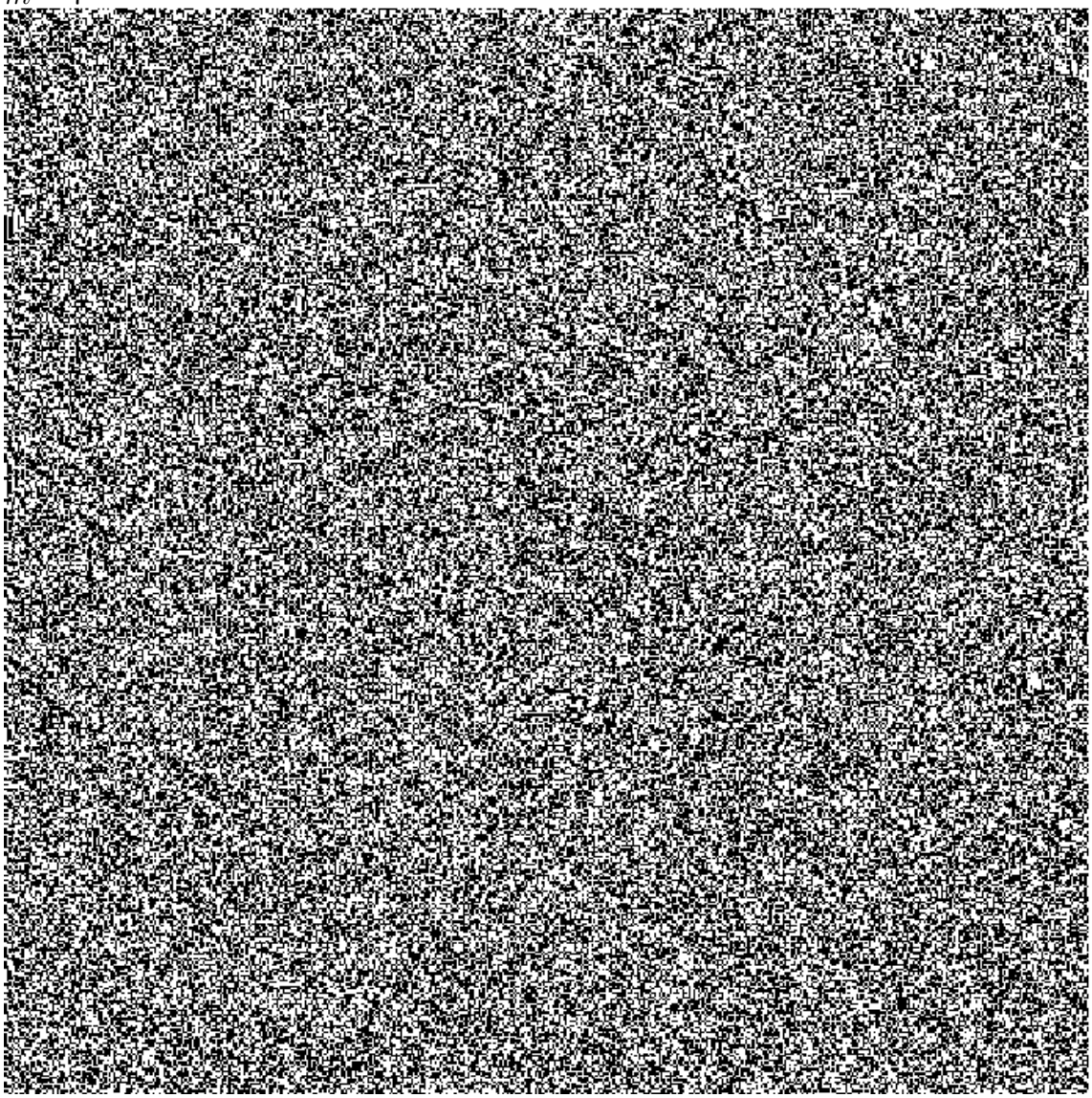


Figura 4.2: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Primeira Execução para $m = 7$



4.1.2 Segunda Execução

Avançando para a segunda execução, a população de candidatos contou com 9 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ onde os dois primeiros elementos vieram da população inicial e as recombinações foram as seguintes: dos índices [1, 0], [1, 2], [3, 2], [1, 4], [2, 5], [6, 5], [7, 4], representando os últimos 6 elementos. Segundo a pontuação que é possível visualizar na tabela 4.4, as *code rules* de forma ordenada são: $\{P_{candidatos}[7], P_{candidatos}[2], P_{candidatos}[1],$
$P_{candidatos}[6], P_{candidatos}[0], P_{candidatos}[8], P_{candidatos}[4], P_{candidatos}[5],$
$P_{candidatos}[3]\}$ sendo que a melhor metade contém os índices $\{7, 2, 1, 6, 0\}$ e a pior os

restantes, $\{8, 4, 5, 3\}$. A tabela 4.5 apresenta os valores dos desvios padrão e a tabela 4.6 as condições fronteira. Note-se quando o número de autómatos disponíveis é ímpar, a melhor metade fica com o elemento do meio.

As *code rules*, em base 32, são:

1. $P_{candidatos}[0] = 2QKJMG2U673QHQSPQAQP569H8V$
2. $P_{candidatos}[1] = 3AE3R8FR0HDU82T2G72KVBVOP8$
3. $P_{candidatos}[2] = 3AC3QO3V2N3U0ACBG62PVF9H9F$
4. $P_{candidatos}[3] = 3AC3U97R2H3U0ASBG62HVFPH9F$
5. $P_{candidatos}[4] = 3AC3QO3V2H7U0ACBG6APVF9H9F$
6. $P_{candidatos}[5] = 3AC3R03V2H7U0ADBG52LVFBP9E$
7. $P_{candidatos}[6] = 3AC3R03V2N7U0ADBG52TVF9H96$
8. $P_{candidatos}[7] = 3AC3R03V2L7S0ADBG52TVFBP96$
9. $P_{candidatos}[8] = 3AC3R2JV2H7S0ACBG5ATVF9H9E$

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.0181	0.0149	0.0113	0.0084	0.0527
$P_{candidatos}[1]$	0.0166	0.0138	0.0106	0.0080	0.0491
$P_{candidatos}[2]$	0.0100	0.0073	0.0061	0.0052	0.0286
$P_{candidatos}[3]$	0.0360	0.0203	0.0156	0.0097	0.0815
$P_{candidatos}[4]$	0.0300	0.0178	0.0109	0.0082	0.0669
$P_{candidatos}[5]$	0.0260	0.0193	0.0147	0.0104	0.0704

Tabela 4.4: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidatos}[x]$	Desvio Padrão
$P_{candidatos}[0]$	0.0284
$P_{candidatos}[1]$	0.0221
$P_{candidatos}[2]$	0.0047
$P_{candidatos}[3]$	0.0104
$P_{candidatos}[4]$	0.0100
$P_{candidatos}[5]$	0.0088
$P_{candidatos}[6]$	0.0069
$P_{candidatos}[7]$	0.0052
$P_{candidatos}[8]$	0.0082

Tabela 4.5: Desvios padrão aproximados para cada $P_{candidato}[x]$

Neste caso, apenas um dos elementos poderá ser considerado para entrar na população ideal final, o elemento $P_{candidatos}[0]$, uma vez que é o único que não apresenta

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[1, 1, 1]
$P_{\text{candidato}}[1]$	[1, 1, 1]
$P_{\text{candidato}}[2]$	[0, 0, 1]
$P_{\text{candidato}}[3]$	[0, 1, 0]
$P_{\text{candidato}}[4]$	[1, 1, 0]
$P_{\text{candidato}}[5]$	[0, 1, 0]
$P_{\text{candidato}}[6]$	[0, 0, 0]
$P_{\text{candidato}}[7]$	[1, 1, 1]
$P_{\text{candidato}}[8]$	[1, 1, 0]

Tabela 4.6: Condições de Fronteira cada $P_{candidato}[x]$

qualquer tipo de padrões ou estruturas, como podemos visualizar na figura 4.3. Como exemplo, temos na figura 4.4 o elemento $P_{candidatos}[1]$ que apresentou um mau resultado, sendo que todos os outros apresentaram um comportamento similar a este.

Figura 4.3: Gráfico da Primeira *Code Rule*, $P_{candidato}[0]$, da Segunda Execução para $m = 7$

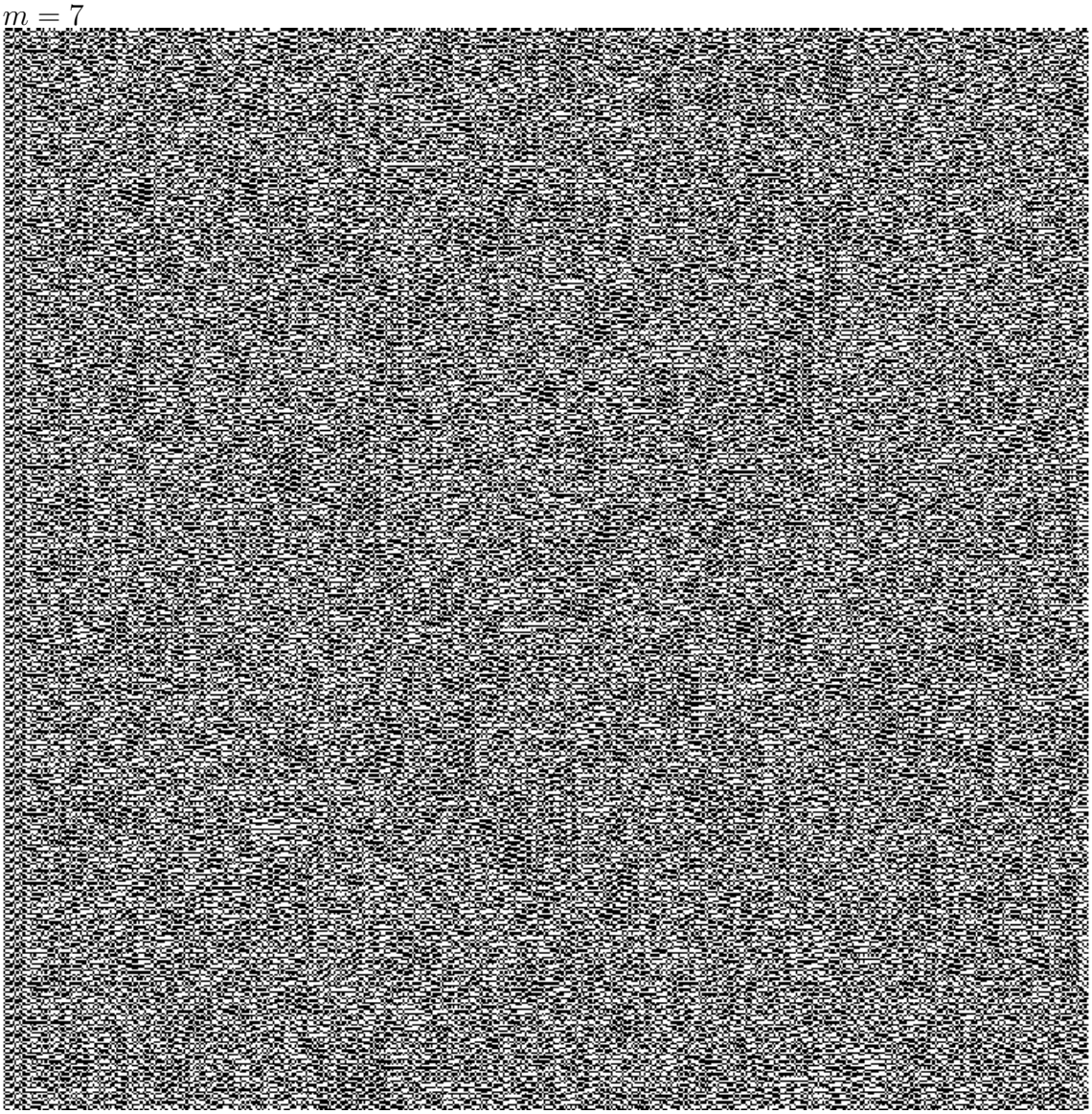
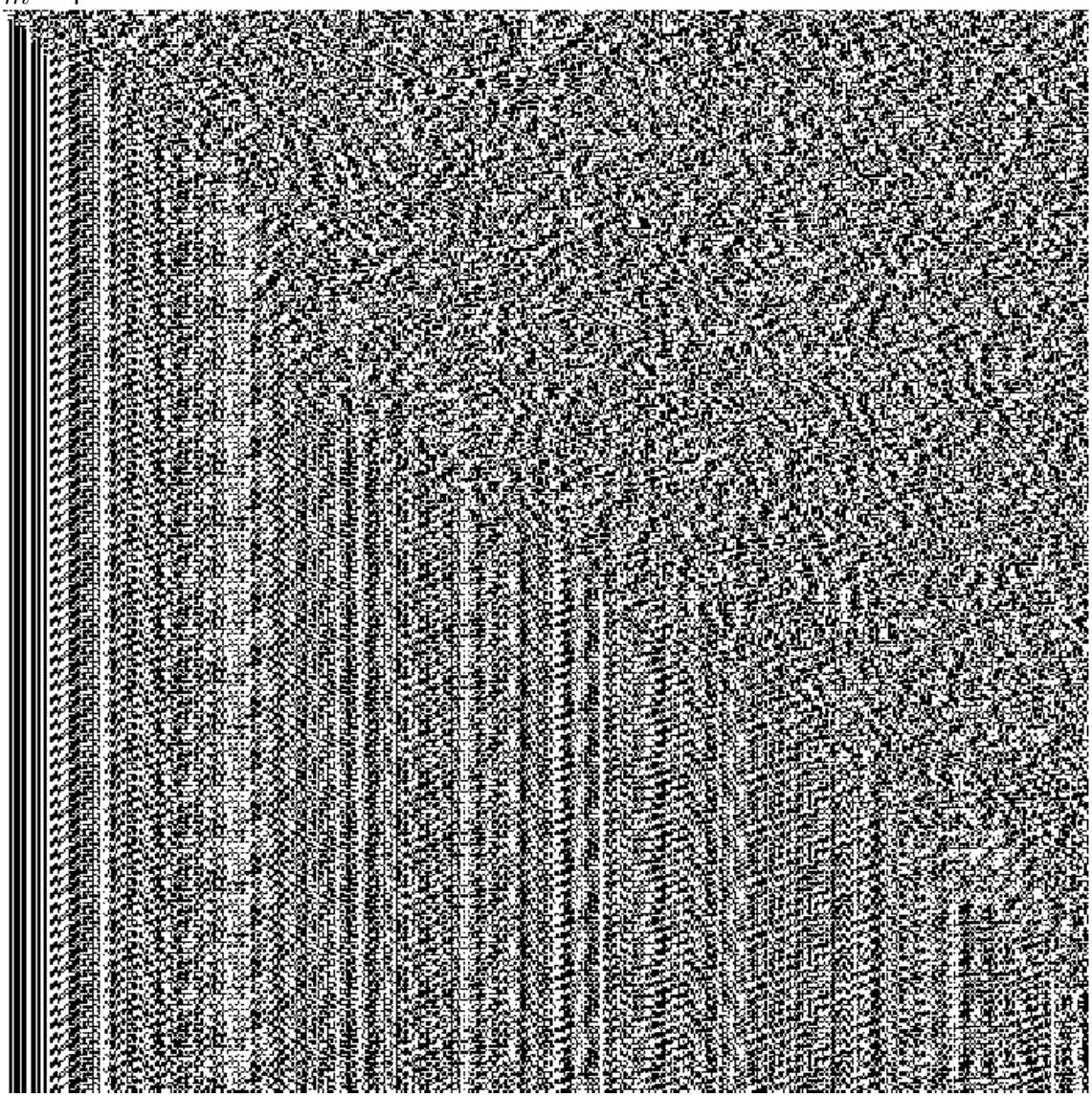


Figura 4.4: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Segunda Execução para $m = 7$



4.1.3 Terceira Execução

A população de candidatos da terceira execução contou com 3 elementos: $P_{candidatos} = \{0, 1, 2\}$ onde os dois primeiros elementos vieram da população inicial e as recombinações foram as seguintes: dos índices [1,0], representando o último elemento. Segundo a pontuação que é possível visualizar na tabela 4.7, as *code rules* de forma ordenada são: $\{P_{candidatos}[2], P_{candidatos}[1], P_{candidatos}[3]\}$ sendo que a melhor metade contém os índices $\{2, 1\}$ e a pior o restante, $\{3\}$. A tabela 4.8 apresenta os valores dos desvios padrão e a tabela 4.9 as condições fronteira.

As *code rules*, em base 32, são:

$P_{candidato}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidato}[0]$	0.0180	0.0150	0.0112	0.0086	0.0527
$P_{candidato}[1]$	0.0169	0.0144	0.0111	0.0083	0.0507
$P_{candidato}[2]$	0.0200	0.0163	0.0137	0.0097	0.0597

Tabela 4.7: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidato}$[x]	Desvios Padrão
$P_{candidato}[0]$	0.026
$P_{candidato}[1]$	0.025
$P_{candidato}[2]$	0.009

Tabela 4.8: Desvios padrão aproximados para cada $P_{candidato}[x]$

1. $P_{candidatos}[0] = 59DVHNFE71MC04S8LJDQ5C49RH$
2. $P_{candidatos}[1] = 3OL8NQ8O8OG469NANL9JPKMVJJ$
3. $P_{candidatos}[2] = 3O5EHQASUOM445M8LNDIR44BRJ$

Dos 3 resultados, 2 deles apresentaram gráficos que demonstram uma aparente aleatoriedade e, como tal, serão incluídas na população ideal final, que se tratam do $P_{candidato}[1]$ e do $P_{candidato}[2]$ como podemos ver nas figura 4.5 e 4.6, respetivamente. O elemento 0 apresenta estruturas e padrões e, como tal, será descartado.

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 1, 0]
$P_{\text{candidato}}[1]$	[1, 0, 0]
$P_{\text{candidato}}[2]$	[0, 1, 0]

Tabela 4.9: Condições de Fronteira cada $P_{candidato}[x]$

Figura 4.5: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Terceira Execução para $m = 7$

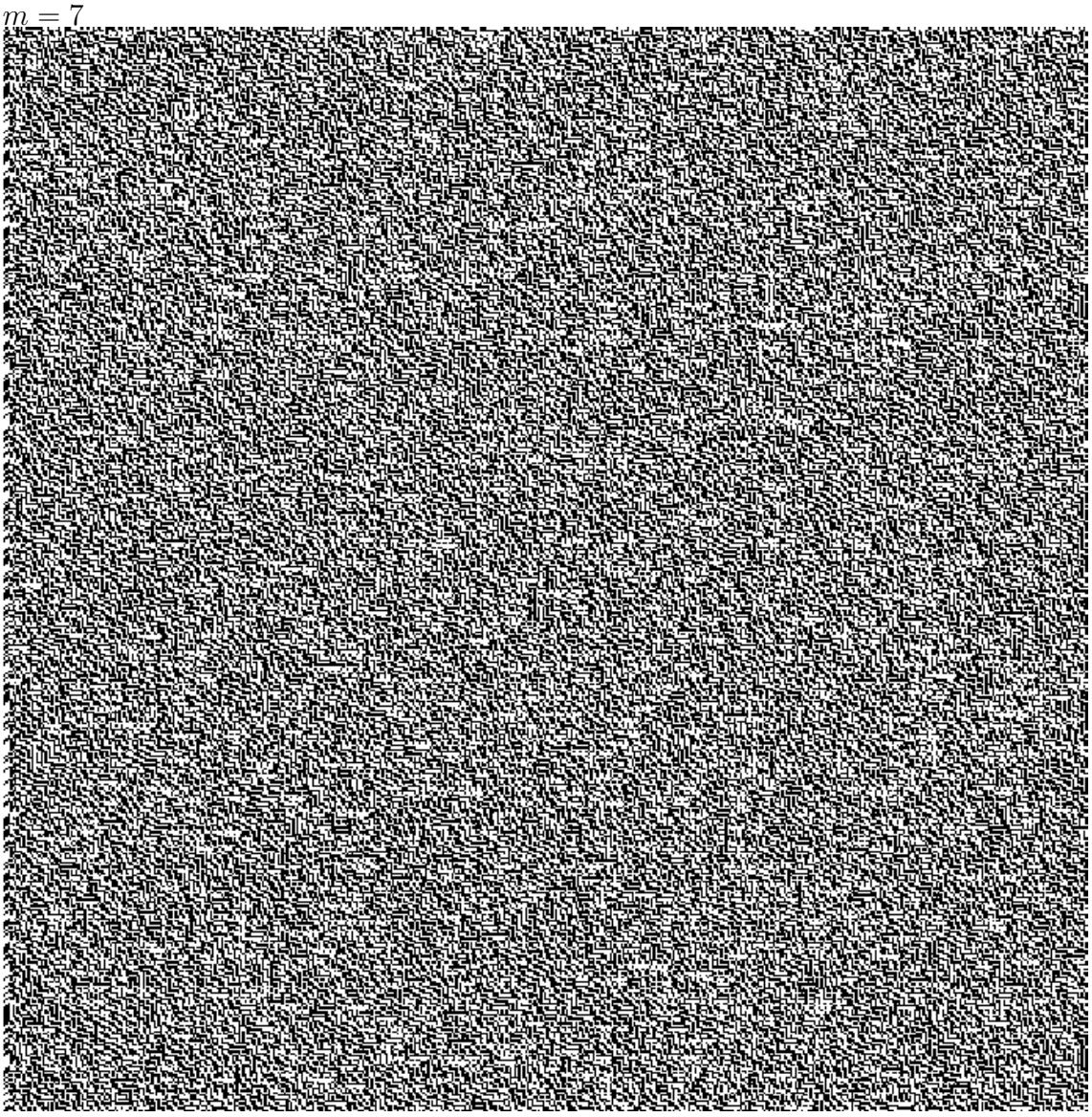
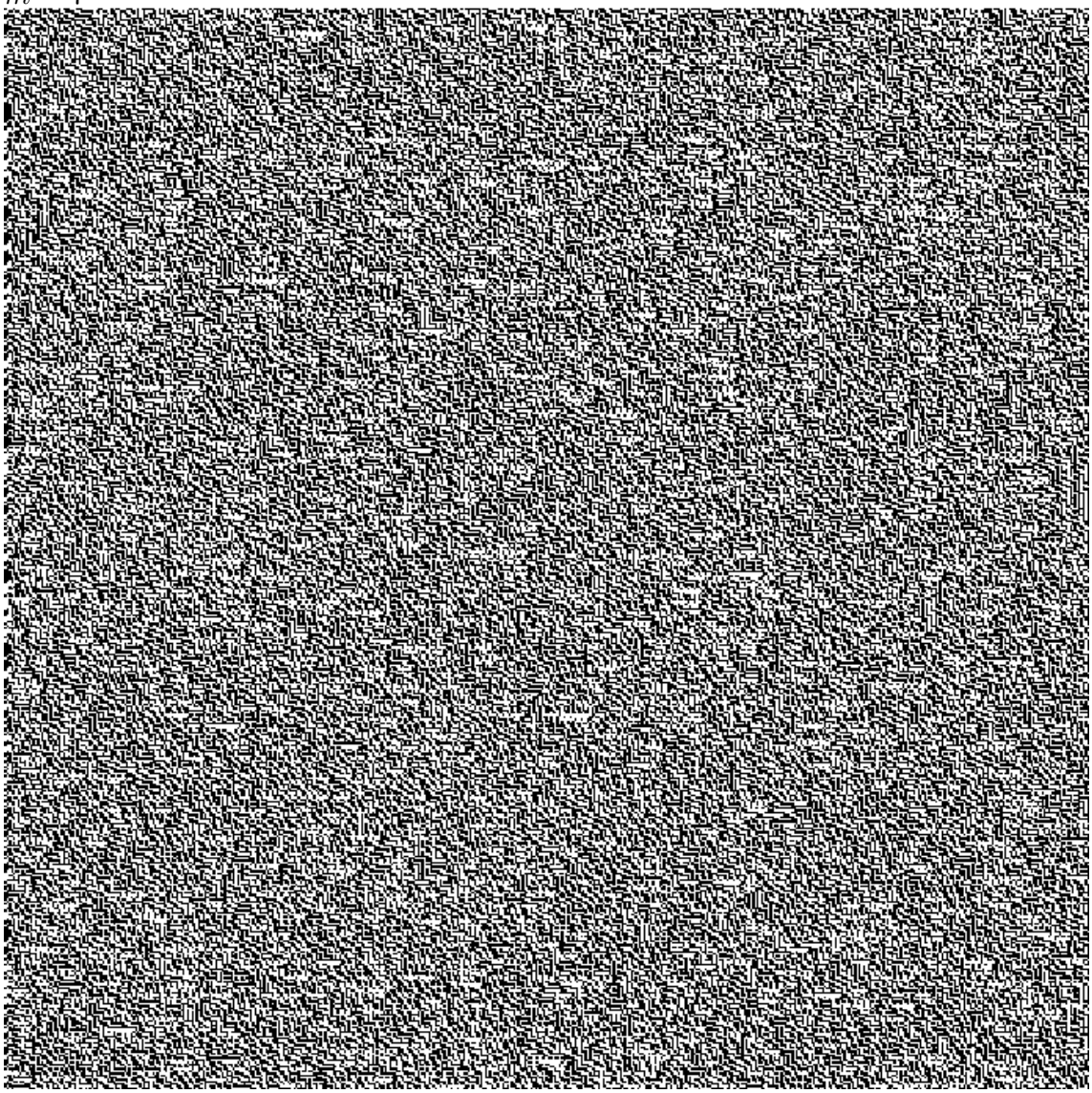


Figura 4.6: Gráfico da Terceira *Code Rule*, $P_{candidato}[2]$, da Terceira Execução para $m = 7$



4.1.4 Quarta Execução

A penúltima execução com $m = 7$, contou com 8 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7\}$ onde os três primeiros elementos vieram da população inicial e as recombinações foram as seguintes: dos índices [1, 0], [2, 3], [2, 0], [3, 0] e [6, 0], representando os 5 últimos elementos. Segundo a pontuação que é possível visualizar na tabela 4.10, as *code rules* de forma ordenada são: $\{P_{candidatos}[3], P_{candidatos}[4], P_{candidatos}[7], P_{candidatos}[6], P_{candidatos}[0], P_{candidatos}[1], P_{candidatos}[2], P_{candidatos}[5]\}$ sendo que a melhor metade contém os índices $\{3, 4, 7, 6\}$ e a pior os restantes, $\{0, 1, 2, 5\}$. A tabela 4.11 apresenta os valores dos desvios padrão

e a tabela 4.12 as condições fronteira.

$P_{candidato}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidato}[0]$	0.0177	0.0145	0.0114	0.0082	0.04477
$P_{candidato}[1]$	0.0181	0.0147	0.0111	0.0084	0.05182
$P_{candidato}[2]$	0.0185	0.0149	0.0112	0.0086	0.07152
$P_{candidato}[3]$	0.0040	0.0075	0.0099	0.0064	0.05321
$P_{candidato}[4]$	0.0040	0.0065	0.0120	0.0087	0.02786
$P_{candidato}[5]$	0.0300	0.0178	0.0156	0.0082	0.03127
$P_{candidato}[6]$	0.0100	0.0165	0.0090	0.0092	0.03410
$P_{candidato}[7]$	0.0120	0.0098	0.0069	0.0054	0.05227

Tabela 4.10: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidato}[x]$	Desvios Padrão
$P_{candidato}[0]$	0.02871
$P_{candidato}[1]$	0.02233
$P_{candidato}[2]$	0.02572
$P_{candidato}[3]$	0.00462
$P_{candidato}[4]$	0.00681
$P_{candidato}[5]$	0.00965
$P_{candidato}[6]$	0.00716
$P_{candidato}[7]$	0.00480

Tabela 4.11: Desvios padrão aproximados para cada $P_{candidato}[x]$

As *code rules*, em base 32, são:

1. $P_{candidatos}[0] = 3OSOD0I2BJM28LV8TKL5FT9T6H$
2. $P_{candidatos}[1] = 59JUJHIK4997RRQBO5IQJLAAG5$
3. $P_{candidatos}[2] = 1SJ6GO2EHN4BSFPQGNB7UG2SKJ$
4. $P_{candidatos}[3] = 3OLS70IMCJ42BVQ9P5MBBTB8I5$
5. $P_{candidatos}[4] = 1SJSN0IE9L4B9VBOH7FB3K2SG7$
6. $P_{candidatos}[5] = 3OS4L02ERI6ASFV8SKI7UL9S6H$
7. $P_{candidatos}[6] = 3OSOD0I6AJM2ATQ9TKKDBTBOIL$
8. $P_{candidatos}[7] = 3OSOD0I2BJM28LV8TKKDBTBSMH$

Neste caso, todos os elementos vão entrar na população devido à sua aparência aleatória, menos os elementos $P_{candidato}[7]$, $P_{candidato}[1]$ e $P_{candidato}[2]$, devido à existência de padrões ou estruturas. Como exemplo, a figura 4.7 apresenta o candidato 0, que foi considerado bom, e na figura 4.8 o candidato 2, que foi considerado mau.

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 1, 1]
$P_{\text{candidato}}[1]$	[0, 0, 1]
$P_{\text{candidato}}[2]$	[0, 0, 0]
$P_{\text{candidato}}[3]$	[1, 1, 1]
$P_{\text{candidato}}[4]$	[1, 1, 1]
$P_{\text{candidato}}[5]$	[0, 1, 1]
$P_{\text{candidato}}[6]$	[0, 0, 1]
$P_{\text{candidato}}[7]$	[0, 0, 0]

Tabela 4.12: Condições de Fronteira cada $P_{candidato}[x]$

Figura 4.7: Gráfico da Primeira *Code Rule*, $P_{candidato}[0]$, da Quarta Execução para $m = 7$

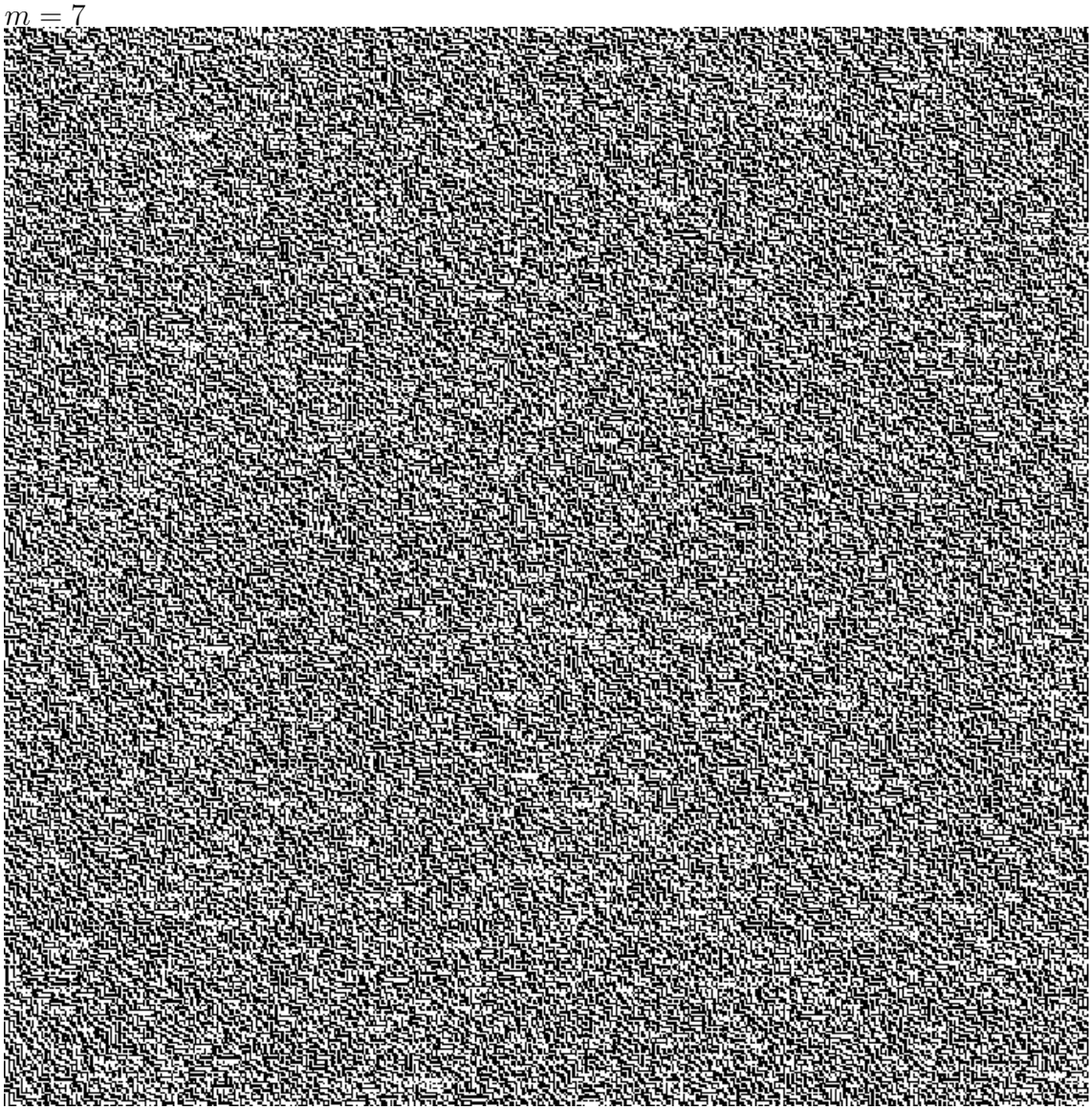
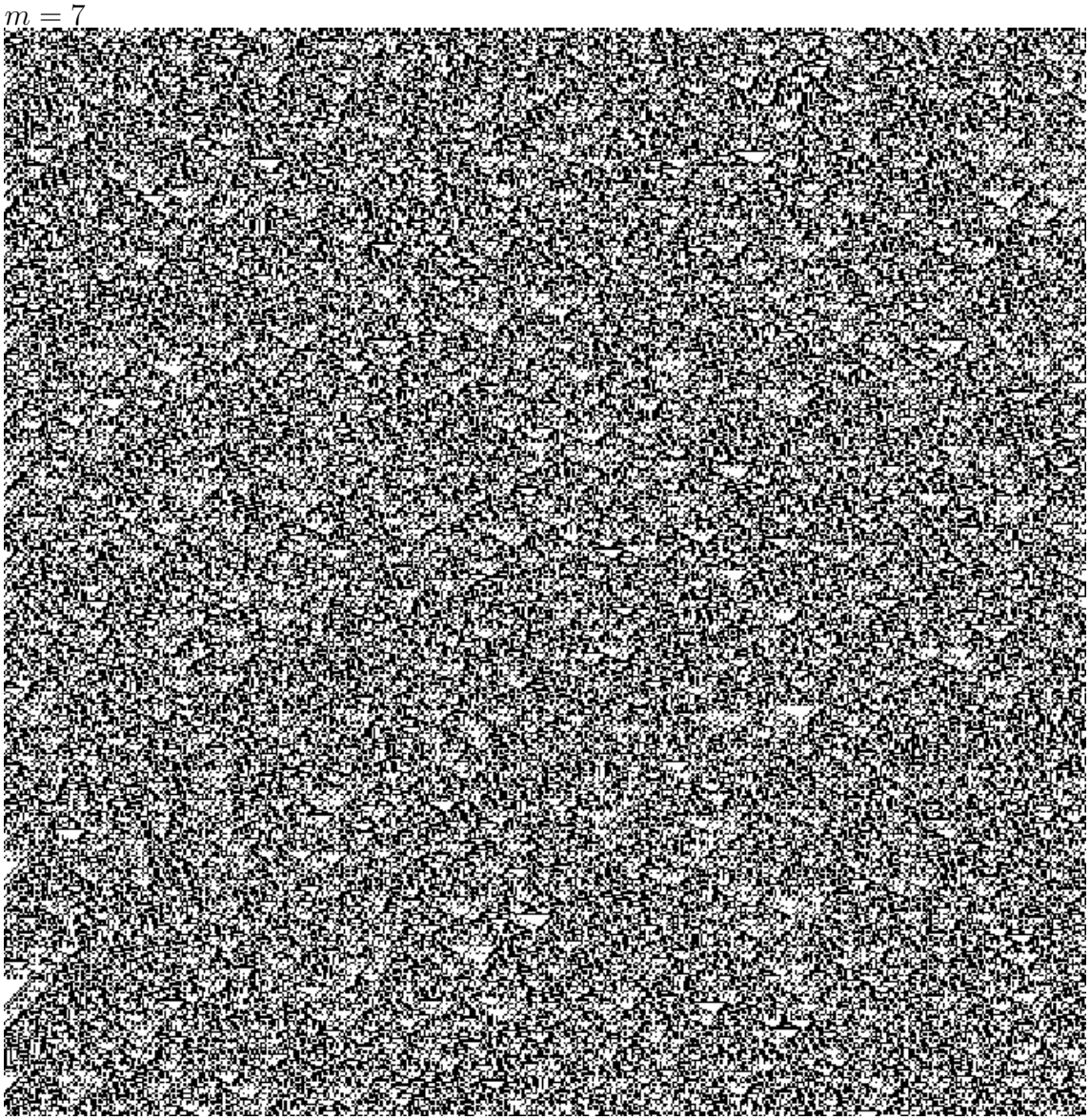


Figura 4.8: Gráfico da Oitava *Code Rule*, $P_{candidato}[7]$, da Quarta Execução para $m = 7$



4.1.5 Quinta Execução

Por fim, a última execução para o $m = 7$ contou com 8 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7\}$ onde os três primeiros elementos vieram da população inicial e as recombinações foram as seguintes: dos índices [0, 1], [3, 0], [2, 0], [0, 4], [5, 0], representando os 5 últimos elementos. Segundo a pontuação que é possível visualizar na tabela 4.13, as *code rules* de forma ordenada são: $\{P_{candidatos}[0], P_{candidatos}[2], P_{candidatos}[1], P_{candidatos}[4], P_{candidatos}[5], P_{candidatos}[3], P_{candidatos}[7], P_{candidatos}[5]\}$ sendo que a melhor metade contém os índices $\{0, 2, 1, 4\}$ e a pior os restantes, $\{5, 3, 7, 5\}$. A tabela 4.14 apresenta os valores dos desvios

padrão.

$P_{candidato}[x]$	Monobits	Bloco 2	Bloco 3	Bloco 4	Resultados
$P_{candidato}[0]$	0.01528	0.01407	0.01060	0.00805	0.04800
$P_{candidato}[1]$	0.01858	0.01492	0.01132	0.00856	0.05339
$P_{candidato}[2]$	0.01722	0.01410	0.01088	0.00798	0.05019
$P_{candidato}[3]$	0.02000	0.01077	0.01556	0.02653	0.07287
$P_{candidato}[4]$	0.02200	0.01152	0.01192	0.00819	0.05364
$P_{candidato}[5]$	0.00800	0.02655	0.03665	0.03413	0.10533
$P_{candidato}[6]$	0.02200	0.01528	0.01242	0.01047	0.06017
$P_{candidato}[7]$	0.03000	0.01879	0.01456	0.02930	0.09265

Tabela 4.13: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidato}[x]$	Desvios Padrão
$P_{candidato}[0]$	0.028
$P_{candidato}[1]$	0.026
$P_{candidato}[2]$	0.026
$P_{candidato}[3]$	0.013
$P_{candidato}[4]$	0.007
$P_{candidato}[5]$	0.024
$P_{candidato}[6]$	0.006
$P_{candidato}[7]$	0.012

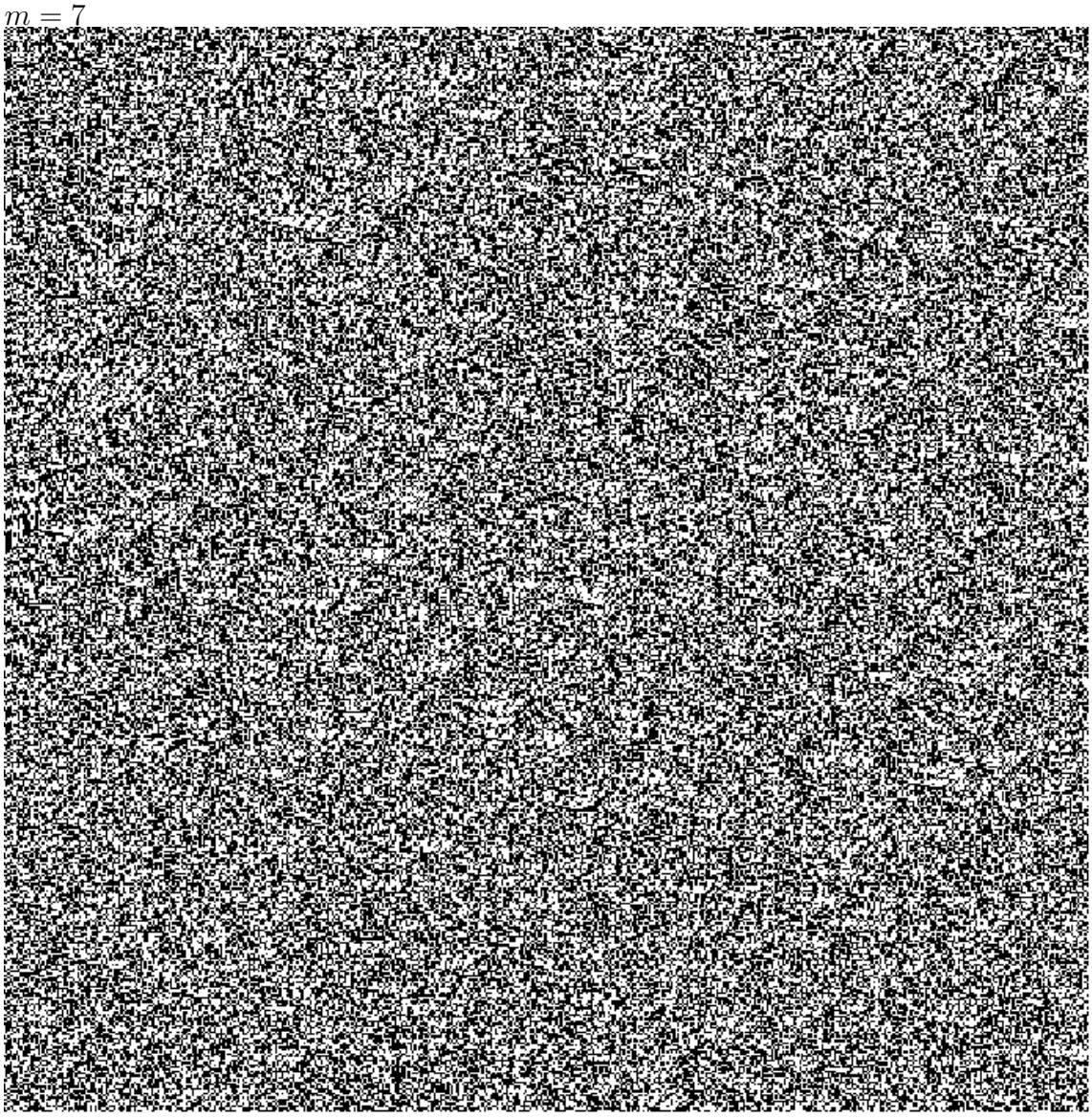
Tabela 4.14: Desvios padrão aproximados para cada $P_{candidato}[x]$

Neste, os elementos que serão admitidos na população ideal final por terem uma aparência aleatória serão os elementos $P_{candidato}[0], P_{candidato}[1], P_{candidato}[4]$, $P_{candidato}[6]$ e o resto será descartado. Como exemplo, na figura 4.9 encontra-se o gráfico do $P_{candidato}[1]$.

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[1, 1, 0]
$P_{\text{candidato}}[1]$	[0, 0, 1]
$P_{\text{candidato}}[2]$	[0, 0, 1]
$P_{\text{candidato}}[3]$	[1, 0, 1]
$P_{\text{candidato}}[4]$	[1, 1, 1]
$P_{\text{candidato}}[5]$	[0, 0, 1]
$P_{\text{candidato}}[6]$	[0, 0, 1]
$P_{\text{candidato}}[7]$	[1, 0, 0]

Tabela 4.15: Condições de Fronteira cada $P_{candidato}[x]$

Figura 4.9: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Quinta Execução para $m = 7$



4.2 Resultados para $m = 9$

Serve esta subsecção para apresentar os resultados das 5 execuções realizadas para $m = 9$. Como já referido anteriormente, tanto para os testes com $m = 9$ como para os restantes, o programa será executado durante 1000 segundos com 100 condições iniciais aleatórias.

4.2.1 Primeira Execução

Na primeira execução foram registados como candidatos, $P_{candidatos}$, para a população ideal final, $P_{idealfinal}$, 3 elementos ao todo, $P_{candidatos} = \{0, 1, 2\}$, onde 2 vieram da população inicial e 1 foi gerado através da recombinação entre os índices [0, 1]. De forma ordenada, $\{P_{candidatos}[1], P_{candidatos}[2], P_{candidatos}[0]\}$

Como as *code rules* são compostas por 512 dígitos, estas serão transformadas em base 32, de forma a ocuparem menos espaço:

1. $P_{candidatos}[0] = 1V9DD0616U4OGQRKBV5BVJ1TAJO2LIBRK3N9LE$
   $17D4JDJMSMAP5N7F83436UJCCDHGNV6918QC65RQ1C7AOOU3B$
   $AGTVHB2VQH4K9Q01$
2. $P_{candidatos}[1] = 2B35V67842TUITJC7J39BP7KG7OBR69BRH07EI8PC$
   $KRLTMJNPJELA2PBPCB0IOFH0CKR66TN59PFQKKB0L7G9KO$
   $V56544QJNLE2HHHM$
3. $P_{candidatos}[2] = V85V06062TUGUB4BN199H7P0JO2R29RP3JF5M9VD$
   $4J5VMSN8HDLFF93H5BQIODL1SNR67D53D55QK4C39EO84GRKV$
   $5L5IJVHCKPRG6$

Utilizando os pontos da tabela 4.16, existiram 2 que foram adicionadas à melhor metade e a última à pior metade. A ordem é a seguinte: $1 \rightarrow 2 \rightarrow 0$. Assim, a melhor metade tem as *code rules*, de forma já ordenada, $\{P_{candidatos}[1], P_{candidatos}[2]\}$ e a pior metade $\{P_{candidatos}[0]\}$. Os desvios padrão podem ser visualizados na tabela 4.17 e as condições fronteira na tabela 4.18.

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.028	0.021	0.016	0.032	0.096
$P_{candidatos}[1]$	0.017	0.015	0.011	0.008	0.052
$P_{candidatos}[2]$	0.022	0.012	0.008	0.015	0.056

Tabela 4.16: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

Na figura 4.10, podemos visualizar o gráfico que foi gerado para a melhor *code rule*, $P_{candidatos}[1]$. Esta *code rule*, apesar de apresentar valores razoáveis através dos cálculos e testes feitos, apresenta padrões e algumas estruturas no gráfico.

De maneira similar à primeira, a segunda melhor *code rule*,$P_{candidatos}[2]$, apresenta também padrões e estruturas. Como tal, nenhum dos elementos da melhor metade apresentou características suficientes para poder ser considerado um **PRNG**.

$P_{candidatos}[x]$	Desvios Padrão
$P_{candidatos}[0]$	0.038
$P_{candidatos}[1]$	0.028
$P_{candidatos}[2]$	0.008

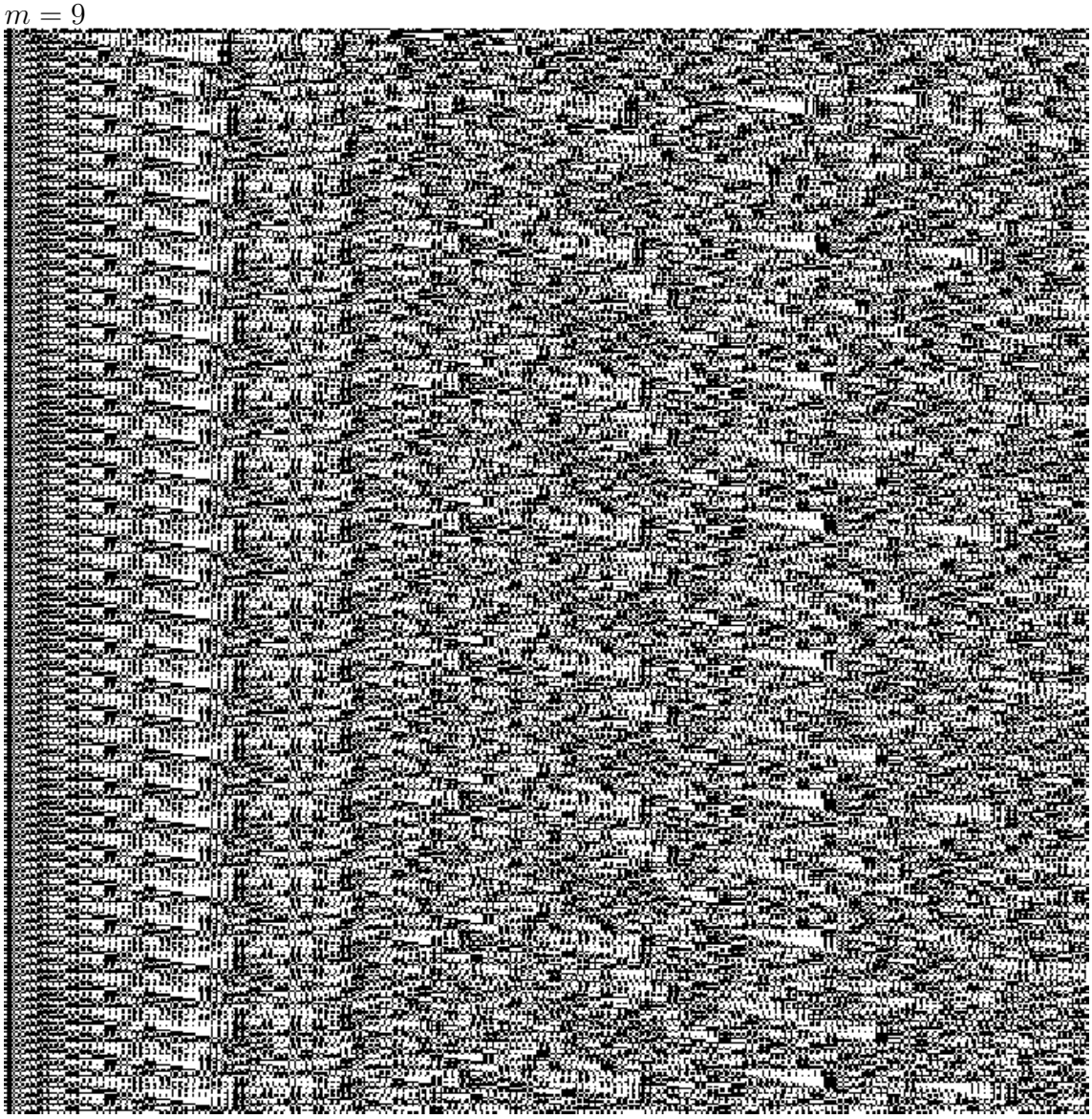
Tabela 4.17: Desvios padrão aproximados para cada $P_{candidato}[x]$

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 0, 1, 1]
$P_{\text{candidato}}[1]$	[1, 1, 1, 1]
$P_{\text{candidato}}[2]$	[1, 1, 1, 0]

Tabela 4.18: Condições de Fronteira cada $P_{candidato}[x]$

Assim, resta-nos apenas 1 *code rule* da pior metade, $P_{candidatos}[0]$, que não obstante às outras duas, também apresenta um gráfico irrelevante. Como tal, nenhum dos casos desta execução será considerado para a população ideal final.

Figura 4.10: Gráfico da Primeira *Code Rule*, $P_{candidato}[0]$, da Primeira Execução para $m = 9$



4.2.2 Segunda Execução

Na segunda execução, a população de candidatos contou com 11 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ e as recombinações foram as seguintes: dos índices [0, 1], [1, 2], [3, 2], [4, 0], [2, 0], [6, 4], [7, 2], [3, 7] e [7, 9], representando os últimos 9 elementos. Segundo a pontuação que é possível visualizar na tabela 4.19, as *code rules* de forma ordenada são: $\{P_{candidatos}[9], P_{candidatos}[3], P_{candidatos}[8], P_{candidatos}[4], P_{candidatos}[1], P_{candidatos}[5], P_{candidatos}[2], P_{candidatos}[7], P_{candidatos}[6], P_{candidatos}[10], P_{candidatos}[0]\}$ sendo que a melhor metade contém os índices $\{9, 3, 8, 4, 1, 5\}$ e a pior os restantes, $\{2, 7, 6, 10, 0\}$.

A tabela 4.20 tem os valores dos desvios padrão e as condições fronteira na tabela 4.21.

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.027	0.019	0.015	0.031	0.092
$P_{candidatos}[1]$	0.016	0.013	0.010	0.008	0.048
$P_{candidatos}[2]$	0.024	0.019	0.015	0.010	0.068
$P_{candidatos}[3]$	0.010	0.009	0.011	0.008	0.038
$P_{candidatos}[4]$	0.010	0.016	0.010	0.008	0.044
$P_{candidatos}[5]$	0.006	0.022	0.014	0.010	0.052
$P_{candidatos}[6]$	0.032	0.016	0.019	0.012	0.079
$P_{candidatos}[7]$	0.022	0.020	0.017	0.011	0.070
$P_{candidatos}[8]$	0.012	0.010	0.011	0.010	0.043
$P_{candidatos}[9]$	0.008	0.005	0.013	0.009	0.035
$P_{candidatos}[10]$	0.036	0.021	0.015	0.011	0.083

Tabela 4.19: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidato}[x]$	Desvio Padrão
$P_{candidato}[0]$	0.039
$P_{candidato}[1]$	0.020
$P_{candidato}[2]$	0.008
$P_{candidato}[3]$	0.005
$P_{candidato}[4]$	0.008
$P_{candidato}[5]$	0.008
$P_{candidato}[6]$	0.011
$P_{candidato}[7]$	0.010
$P_{candidato}[8]$	0.008
$P_{candidato}[9]$	0.006
$P_{candidato}[10]$	0.011

Tabela 4.20: Desvios padrão aproximados para cada $P_{candidato}[x]$

As *code rules*, em base 32, são as seguintes:

1. $P_{candidatos}[0] = 1FDMQDI836JFCBA3KIF1TTOH99B52BV6257C6Q$
   $OTOLO4FNMAFV8K47SG1885P8FKDN7G4F9U6F4E8SMCFR32$
   $K06VSCPHD5AKOMBLER6$

2. $P_{candidatos}[1] = 397SAL50NK38PQPUTGCGP7UVD9SR240K34QSC$
   $D6AOENIUDCFPI4D991TP9C8CODR5AIMRHSFQETO9CHRG9$
   $B00NR137F383OLPV7HF39$

3. $P_{candidatos}[2] = 1DFKQ5U8V63DOROMTGCPP7UR99ER2AD635AE$
   $CB9SOCQ0VNMEBUCC018GP885COFJ5U6K01PV6EKS9SKP6P$
   $B04LQ5O6V3C3AKOV7HEJF$

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 0, 1, 1]
$P_{\text{candidato}}[1]$	[1, 0, 0, 0]
$P_{\text{candidato}}[2]$	[0, 1, 0, 0]
$P_{\text{candidato}}[3]$	[1, 1, 1, 0]
$P_{\text{candidato}}[4]$	[0, 1, 0, 0]
$P_{\text{candidato}}[5]$	[0, 0, 0, 1]
$P_{\text{candidato}}[6]$	[0, 1, 1, 0]
$P_{\text{candidato}}[7]$	[0, 1, 1, 0]
$P_{\text{candidato}}[8]$	[0, 1, 0, 0]
$P_{\text{candidato}}[9]$	[0, 0, 0, 0]
$P_{\text{candidato}}[10]$	[0, 0, 1, 0]

Tabela 4.21: Condições de Fronteira cada $P_{candidato}[x]$

4. $P_{candidatos}[3] = 1DFKQLU8NK39ORPQTGCHP7UV994R20C624AE$
$CDEAOCM0UN6EBU4C190SP9C9COFJ1U6MPHTFAELS9CLRI$
$PB04NQ137F383ALPV7HEJ9$

5. $P_{candidatos}[4] = 1DFKQ5U8V63DOROQTGCJP7UV99ER20C625AE$
$SDEOOCI0UN6EBU4C098OP8C1COFJ5U6K1HPV2EKS9CLRM$
$PB04NR1G7V383ALPV7HEJB$

6. $P_{candidatos}[5] = 1DDMQDM8F6JF8ROIS0C1TTUH99EA29V6212E6C$
$OOOTQ4FNMEFV044BOG98C5COFM5M4K1B9V2B0C8SNRVPB2$
$K1RHG5RHD7ALONFLERF$

7. $P_{candidatos}[6] = 1DDKGDI8V6JDSBQ7LGE9P7SP99B92AT6352CEA$
$9TOTO4FNMEFFCC41SGP881POFH5V6K07PM6E4C9SM9ER324$
$4UVO6TJC7AKOF3HERF$

8. $P_{candidatos}[7] = 1DFKGDM8V6JDOBQILGCPP7SP99FR2AD625AE$
$CCETOSQ8EN6EFU4C41OOP8818OFJ5U6K13PMEEKC9SKB7P3$
$04KUPO7VJC7AKOV3HERF$

9. $P_{candidatos}[8] = 1DFKIDQ8V6JDOBQITGCPP7UR99FR2AD635AE$
$CAETOCQ0VN6EFUCC018GP881COFJ5S6K03PUEEKS9SK96P3$
$04KUPO7V3C3AKOV7HEJF$

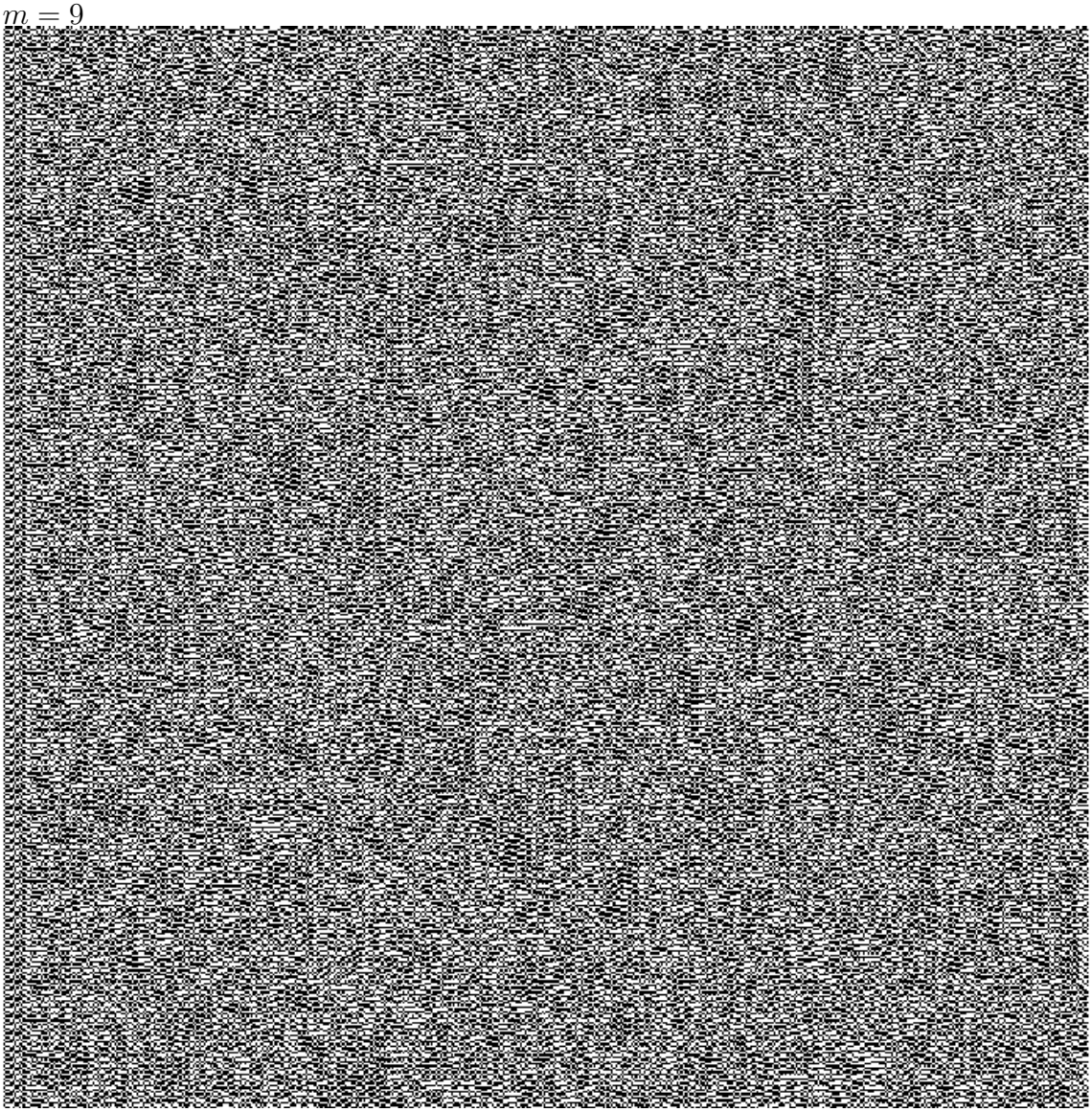
10. $P_{candidatos}[9] = 1DFKOLU8N63DOBOQTGCHP7SV995R20C625AE$
$CDEROSQ8EN6EFU4C51OSP8C98OFJ5U6K93T6AELC9CLRIP3$
$04MQ1R7FJ87AKPV3HER9$

11. $P_{candidatos}[10] = 1DFKODU8N6JDOBOILGCHP7SR995R28C625AE$
$CDEVOSQ8EN6EFU4C51OOP8898OFJ5U6K93P6EELC9CLR2P3$
$04MU1R7VJ87AKSV3HERD$

Nesta execução, apenas um dos resultados apresenta um gráfico digno de ser integrado na população ideal final, que é o $P_{candidato}[1]$. O aspeto deste, como podemos

ver na figura 4.11, foi o que apresentou aleatoriedade aparente, ao contrário de todos os outros, sendo que ficaram muito parecidos entre eles, com bastantes padrões no lado esquerdo.

Figura 4.11: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Segunda Execução para $m = 9$



4.2.3 Terceira Execução

Avançando para a terceira execução, a população ideal contou com 11 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, onde os dois primeiros vieram da população inicial e os restantes através da recombinação dos índices [0, 1] [0, 2], [3, 1], [3, 0], [3, 4], [5, 3], [1, 7], [4, 2] e [4, 7]. Seguindo a pontua-

ção da tabela 4.22, a melhor metade tem as *code rules*, de forma já ordenada, $\{P_{candidatos}[9], P_{candidatos}[6], P_{candidatos}[3], P_{candidatos}[2], P_{candidatos}[0], P_{candidatos}[1]\}$ e a pior metade $\{P_{candidatos}[4], P_{candidatos}[5], P_{candidatos}[7], P_{candidatos}[10], P_{candidatos}[8]\}$. A tabela 4.23 apresenta os valores dos desvios padrão e as na tabela 4.24 as condições fronteira.

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.018	0.014	0.011	0.009	0.052
$P_{candidatos}[1]$	0.017	0.015	0.012	0.009	0.053
$P_{candidatos}[2]$	0.014	0.016	0.012	0.009	0.051
$P_{candidatos}[3]$	0.002	0.013	0.010	0.008	0.032
$P_{candidatos}[4]$	0.022	0.017	0.011	0.009	0.059
$P_{candidatos}[5]$	0.014	0.024	0.014	0.008	0.059
$P_{candidatos}[6]$	0.010	0.006	0.008	0.006	0.030
$P_{candidatos}[7]$	0.028	0.020	0.010	0.010	0.069
$P_{candidatos}[8]$	0.032	0.025	0.016	0.013	0.086
$P_{candidatos}[9]$	0.006	0.003	0.004	0.003	0.016
$P_{candidatos}[10]$	0.026	0.022	0.019	0.011	0.079

Tabela 4.22: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidatos}[x]$	Desvios Padrão
$P_{candidatos}[0]$	0.025
$P_{candidatos}[1]$	0.025
$P_{candidatos}[2]$	0.008
$P_{candidatos}[3]$	0.006
$P_{candidatos}[4]$	0.007
$P_{candidatos}[5]$	0.011
$P_{candidatos}[6]$	0.006
$P_{candidatos}[7]$	0.011

Tabela 4.23: Desvios padrão aproximados para cada $P_{candidato}[x]$

As *code rules*, em base 32, são as seguintes:

1. $P_{candidatos}[0] = 31ITI525BROBIHMSCA829RESFNIVN6K3ASB2C$
   $QUPO0AILM2JVTIGB0JC3IDHLLN1GMKFIPOJ11F88SNCRP$
   $MI3V8J963GU54B7TS2REJ$
2. $P_{candidatos}[1] = 31QAGNTSQ5F5Q8N1DN1M1BCEGU9NOMPT0UK$
   $TU1QIMOR419E67BH06U9I2TQP2E5J76PKS87AVGUVP0V3G4LQ$
   $91BTR5521SJF66IGMQK$
3. $P_{candidatos}[2] = 31QQI76DB3C3Q8MTDN9M1RCCHN3NKUSD1UO$
   $SSAQQSCQ01OEJNDI0A4J83ICP2EN1M6O7O8O2DGVFASV5PC$
   $NQ39BPV5321L6B7DM2IEM$

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 0, 1, 0]
$P_{\text{candidato}}[1]$	[1, 0, 0, 1]
$P_{\text{candidato}}[2]$	[1, 0, 1, 1]
$P_{\text{candidato}}[3]$	[0, 0, 0, 1]
$P_{\text{candidato}}[4]$	[1, 0, 0, 0]
$P_{\text{candidato}}[5]$	[1, 1, 0, 1]
$P_{\text{candidato}}[6]$	[0, 1, 1, 0]
$P_{\text{candidato}}[7]$	[1, 1, 0, 0]
$P_{\text{candidato}}[8]$	[1, 1, 1, 1]
$P_{\text{candidato}}[9]$	[1, 1, 1, 1]
$P_{\text{candidato}}[10]$	[1, 0, 0, 0]

Tabela 4.24: Condições de Fronteira cada $P_{candidato}[x]$

4. $P_{candidatos}[3] = 31QOI76DBRC3I1MTCI8I9RCC3NJNN6K9AS8USQ$
$QPSCQ25OEJNDIGB0J83IDH66N1I6GFGOO3D0VB8SN5RCNQ3$
$R9JV43GF54B7TS2REM$

5. $P_{candidatos}[4] = 31QQI7NCBNF5I1MLCI0M9BCCINBNTMGD2UKS$
$UBQGU8Q45PEJ7FIG369O2OTH2E71I6PUSOUAD0UF9KN5J4NQ9$
$33HR47I15MF7NI0RAG$

6. $P_{candidatos}[5] = 31ISI725BRO3IHMTC28I9RES7NINN6KBASA6SQQ$
$PSCAI5I2JVDIGB0JC3IDHKMN1I6GFGOOJ5HFA8SNDRPMI3R$
$9JD63GU54B7TS2RFI$

7. $P_{candidatos}[6] = 31QQI76CBNF7I9MLCI0I9RCC3NBNNMG92U8UUQ$
$QPSCQ25OEJNFIGB2P83ITH2671I6GEGOSAD0VB8SN5J4NQ9R9$
$JR47G556F7TQ0REI$

8. $P_{candidatos}[7] = 31IOI72DBRS3IHMTC2AI9RES3NINN6K9ASA6SQQ$
$PSCAI5I2JNDIKB0JC3IDHM6N1I6GFGOO351FA8SNDV9MI3R9J$
$T43KU54BNTS2REM$

9. $P_{candidatos}[8] = 31Q8G71TQFF5I9N5DM3I9REC0UBNSMS90SOLU0Q$
$JUCAI1BEM7DJK7EHU2MCPI6N12MHSKOT27HEA98V3NCKI1R$
$PHP41IHLLBN6PGNAM$

10. $P_{candidatos}[9] = 31QQIN6CB7C1I8MLDJ8M1RCCIN3NTMOD0UOSU$
$BQOSCQ49PEJ7FIG34B83IDR2E71M6PNSOQ2D0VF8LV5J4NQ13B$
$HV572156B75M2JEK$

11. $P_{candidatos}[10] = 31IQI6JDBRC1I1MLC2AM9BESJNRNVMK9AUIM$
$UBQ1S8A05J2JNDIGB0RC3QDHIE71I6PESOUB50EE8SN5R5N2131$
$JV47GR56BNTG0REM$

Os resultados desta execução parecem, à primeira vista, promissores. No entanto, a parte do meio e da direita do gráfico parece aleatório em alguns dos resultados, mas

o lado esquerdo estraga a possibilidade de poderem ser considerados relevantes para a população ideal final. Dois exemplos desta execução encontram-se na figura 4.5, que apresenta o elemento 0, e na figura 4.13, que apresenta o elemento 2.

Figura 4.12: Gráfico da Primeira *Code Rule*, $P_{candidato}[0]$, da Terceira Execução para $m = 9$

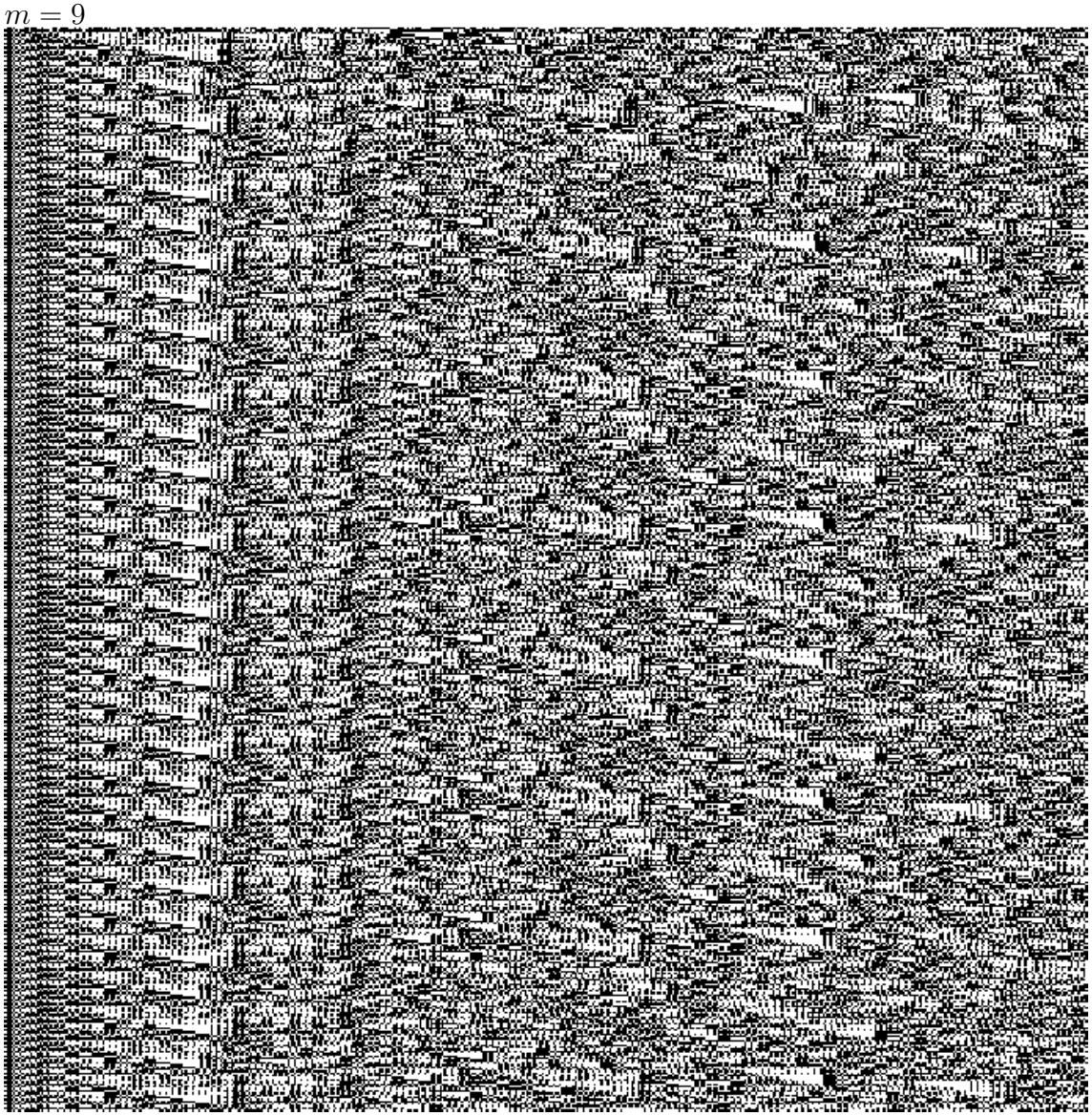
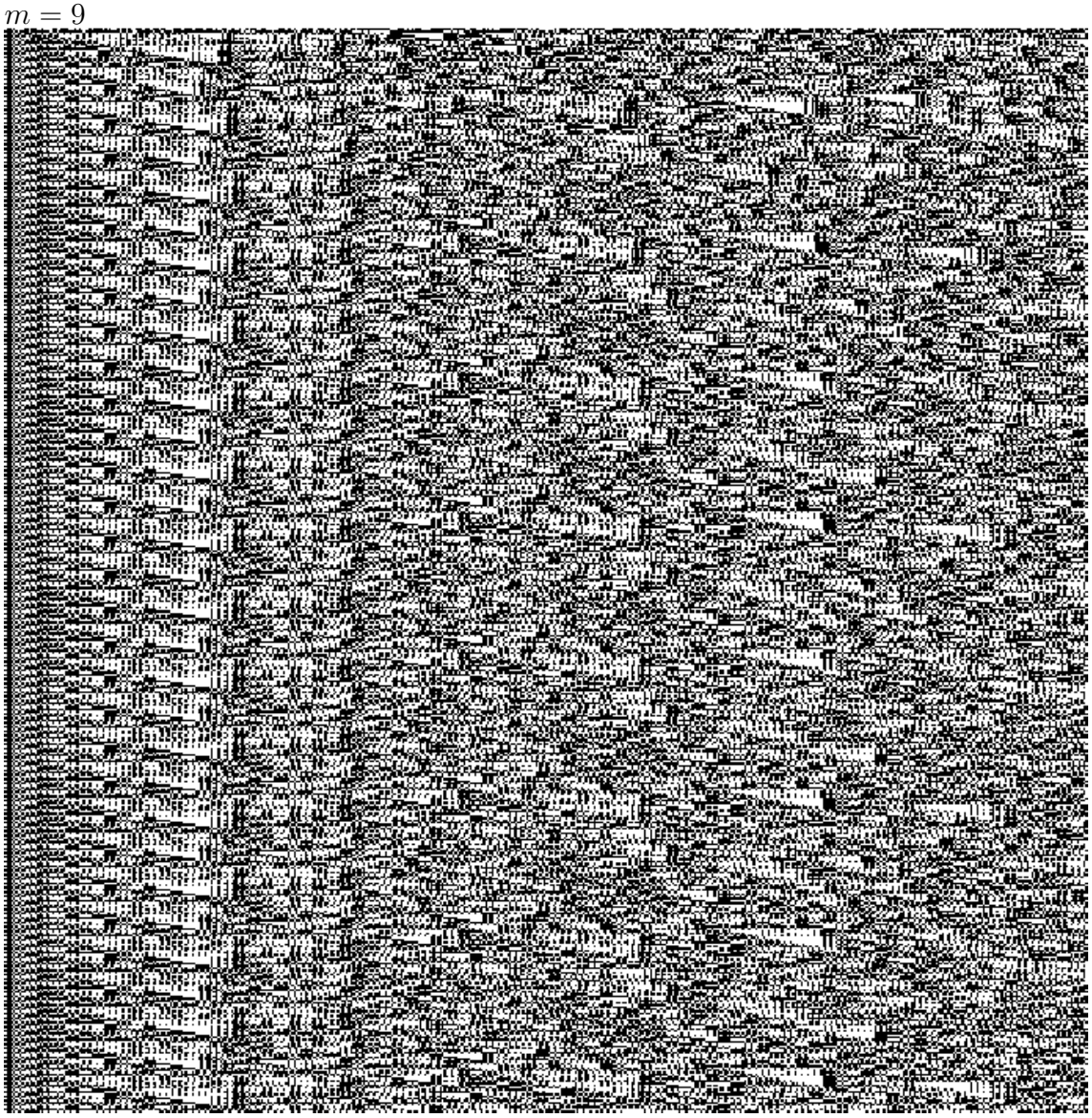


Figura 4.13: Gráfico da Terceira *Code Rule*, $P_{candidato}[2]$, da Terceira Execução para $m = 9$



4.2.4 Quarta Execução

Após realizar a quarta execução, a população ideal contou com 6 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5\}$, onde os três primeiros vieram da população inicial e os restantes através da recombinação dos índices [1, 0], [1, 2], [3, 1], [3, 2]. Através da tabela 4.25, a melhor metade tem as *code rules*, de forma já ordenada, $\{P_{candidatos}[4], P_{candidatos}[5], P_{candidatos}[1]\}$ e a pior metade $\{P_{candidatos}[0], P_{candidatos}[3], P_{candidatos}[2]\}$. A tabela 4.26 apresenta os valores dos desvios padrão de cada um e a tabela 4.27 as condições fronteira.

As *code rules*, em base 32, são as seguintes:

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.019	0.016	0.012	0.009	0.055
$P_{candidatos}[1]$	0.018	0.014	0.011	0.008	0.051
$P_{candidatos}[2]$	0.038	0.026	0.020	0.017	0.101
$P_{candidatos}[3]$	0.028	0.015	0.011	0.009	0.063
$P_{candidatos}[4]$	0.016	0.010	0.009	0.006	0.042
$P_{candidatos}[5]$	0.020	0.011	0.011	0.007	0.049

Tabela 4.25: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidatos}[x]$	Desvios Padrão
$P_{candidatos}[0]$	0.025
$P_{candidatos}[1]$	0.027
$P_{candidatos}[2]$	0.012
$P_{candidatos}[3]$	0.009
$P_{candidatos}[4]$	0.006
$P_{candidatos}[5]$	0.006

Tabela 4.26: Desvios padrão aproximados para cada $P_{candidato}[x]$

1. $P_{candidatos}[0] = 15TJ5QNOPHEGMK5GLG1DM7I419G7JJ9AIUM94$
$S6VBUPL3EF76GLMN7VICP4CO5MIIKL0KRRVAIM1ELKK7T$
$VJS2BHUJ1HJNJG808FIAA$
2. $P_{candidatos}[1] = 3O8CKCQ2BGTHF8BBP752AFAUMET6F46GLPG3$
$2MTE52A6J10531UUK84NBURND2N0TSJS26PALKC0VAVVKGO$
$5P4KQMJDQ5GPRDQMPJNI$
3. $P_{candidatos}[2] = 39DCL8V0PHTHF8B9HI1CM7I44BO7F590KSM32K$
$DE1Q9LJ7D561SUN2KIFR5TD4M2RKL02VRASGE1E5VL5SVJU0$
$DJUJ9G3IRP9QSBIMI$
4. $P_{candidatos}[3] = 3OCCK8U2PHTHF8B9HM5EAF2M4AS6F590KTK3$
$2KDE529KI3C531UUK84NFRTVD4M0TSHS2NRATKC0EAVV1SP$
$3O4LJUJ9I5GPR9QMRJNI$
5. $P_{candidatos}[4] = 3OCCKCU2PGTHF8B9H75AQF2MMAS6F410LTK3$
$2KTE5284I18531UUK84NBURND4M0TUJS27PATKC0VAVVKSO7$
$P4LIUNDI5GPR9QMPJNI$
6. $P_{candidatos}[5] = 39DCL8U2PHTHF8B9HM1CI7I64BO7F590KTK32K$
$DE1A9KJ7C561SUN2KJFRLVDKM0TSL02VRASGE0E5VL1ST3S$
$05JUJ9G7ITR9QSBJNI$

Dos 6 resultados obtidos, apenas o $P_{candidato}[2]$ e o $P_{candidato}[5]$ apresentaram aparente aleatoriedade nos respetivos gráficos, enquanto que os outros apresentaram padrões. Como tal, serão inseridos na população ideal final. Nas figuras 4.7 e 4.8 encontram-se estes 2 elementos, respetivamente.

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[1, 1, 0, 1]
$P_{\text{candidato}}[1]$	[1, 0, 0, 0]
$P_{\text{candidato}}[2]$	[1, 0, 0, 1]
$P_{\text{candidato}}[3]$	[1, 0, 1, 0]
$P_{\text{candidato}}[4]$	[1, 1, 0, 0]
$P_{\text{candidato}}[5]$	[1, 1, 1, 0]

Tabela 4.27: Condições de Fronteira cada $P_{candidato}[x]$

Figura 4.14: Gráfico da Terceira *Code Rule*, $P_{candidato}[2]$, da Quarta Execução para $m = 9$

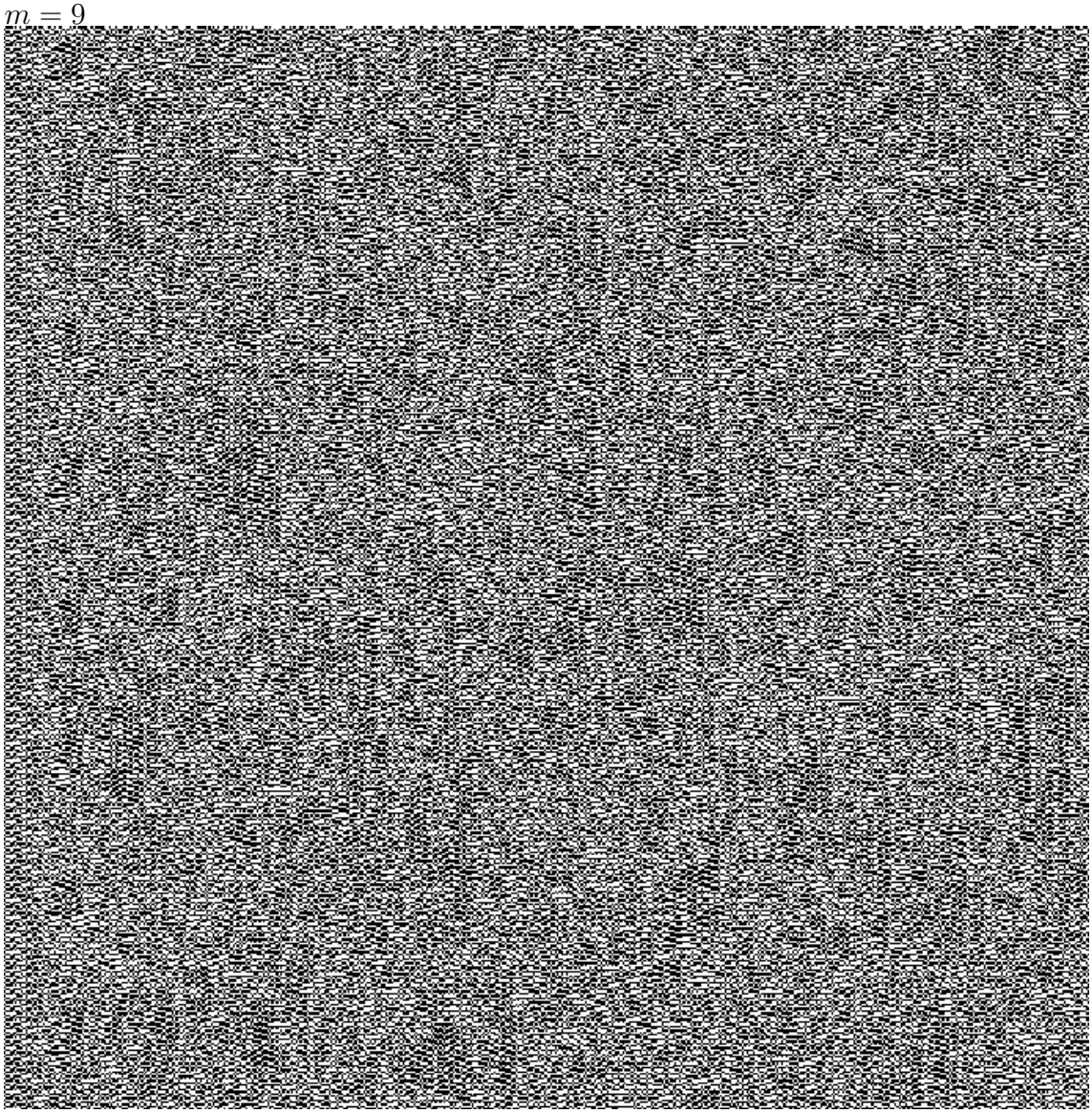
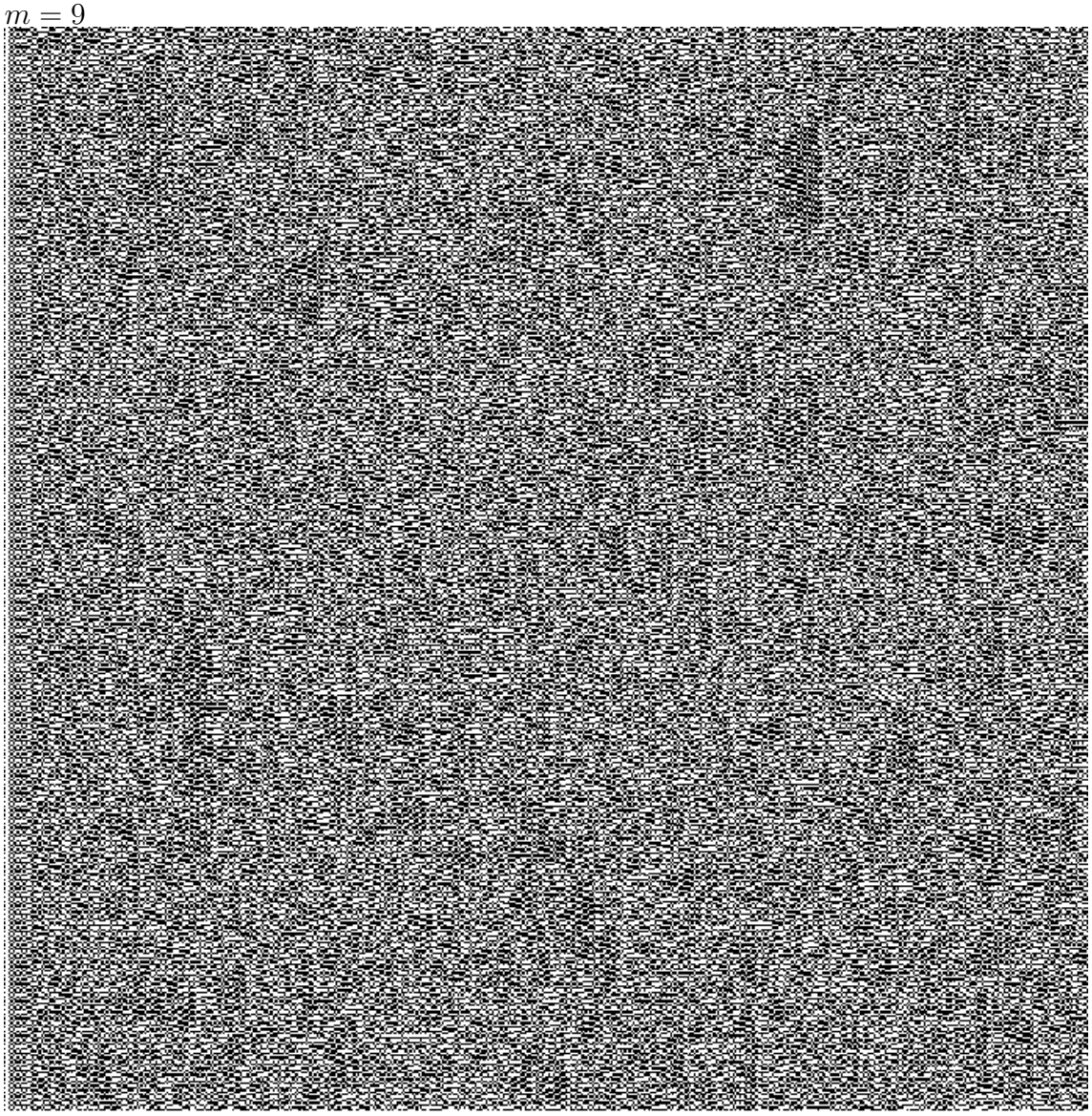


Figura 4.15: Gráfico da Sexta *Code Rule*, $P_{candidato}[5]$, da Quarta Execução para $m = 9$



4.2.5 Quinta Execução

Por fim, será realizada a quinta execução. Neste caso, a população ideal contou com 12 elementos: $P_{candidatos} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$, onde os três primeiros vieram da população inicial e os restantes através da recombinação dos índices [2, 0], [0, 3], [4, 2], [4, 0], [4, 5], [7, 5], [8, 0], [7, 0]. Pela tabela 4.28, a melhor metade tem as *code rules*, de forma já ordenada, $\{P_{candidato}[6], P_{candidato}[8], P_{candidato}[2], P_{candidato}[1], P_{candidato}[0], P_{candidato}[4]\}$ e a pior metade $\{P_{candidato}[3], P_{candidato}[7], P_{candidato}[9], P_{candidato}[10], P_{candidato}[5]\}$. A tabela 4.29 apresenta os desvios padrão e a tabela 4.30 as condições fronteira.

$P_{candidatos}[x]$	*Monobits*	Bloco 2	Bloco 3	Bloco 4	Resultado
$P_{candidatos}[0]$	0.0185	0.0148	0.0114	0.0083	0.0530
$P_{candidatos}[1]$	0.0169	0.0139	0.0106	0.0082	0.0497
$P_{candidatos}[2]$	0.0156	0.0141	0.0108	0.0084	0.0489
$P_{candidatos}[3]$	0.0280	0.0163	0.0120	0.0089	0.0652
$P_{candidatos}[4]$	0.0240	0.0133	0.0136	0.0101	0.0610
$P_{candidatos}[5]$	0.0400	0.0213	0.0141	0.0087	0.0840
$P_{candidatos}[6]$	0.0000	0.0045	0.0070	0.0061	0.0177
$P_{candidatos}[7]$	0.0280	0.0158	0.0142	0.0102	0.0681
$P_{candidatos}[8]$	0.0180	0.0095	0.0097	0.0079	0.0451
$P_{candidatos}[9]$	0.0300	0.0147	0.0136	0.0120	0.0703
$P_{candidatos}[10]$	0.0360	0.0178	0.0090	0.0104	0.0733

Tabela 4.28: Valores aproximados das médias para cada $P_{candidato}[x]$ e resultados

$P_{candidatos}[x]$	Desvios Padrão
$P_{candidatos}[0]$	0.0262
$P_{candidatos}[1]$	0.0279
$P_{candidatos}[2]$	0.0234
$P_{candidatos}[3]$	0.0094
$P_{candidatos}[4]$	0.0075
$P_{candidatos}[5]$	0.0133
$P_{candidatos}[6]$	0.0043
$P_{candidatos}[7]$	0.0094
$P_{candidatos}[8]$	0.0065
$P_{candidatos}[9]$	0.0108
$P_{candidatos}[10]$	0.0126

Tabela 4.29: Desvios padrão aproximados para cada $P_{candidato}[x]$

As últimas *code rules* criadas pelas execuções de $m = 9$, em base 32, são as seguintes:

1. $P_{candidatos}[0] = U570V69IBJN3A2FO03F6IC6LT8J0CGT9BF5BMOH$
   $VQ9DFE60NGKRICNMBN07FO31ISQT85N0GCLMUTDML7B9F$
   $U09D64GG2N25FUPPRO$
2. $P_{candidatos}[1] = 3905IM0G9OOO7L7BTGVAIV6NDEVH380OAFIMN7$
   $LKG40S3KNIUQ2B27A74CD6A23EBBBS057DJJQK9BCNSLQDQS$
   $FCEUNE4VBC9H1CREA$
3. $P_{candidatos}[2] = MN4KNJ3G277MG26KGOO1S94CE2GDPDMVMK4$
   $C25HP7GD37EFAVI2U4MM1E9LJR6SNVQ9GTR76DULLLNTR7T$
   $9I29PNQ93IKI6E4744JD$
4. $P_{candidatos}[3] = MN5KV21GAJN3O27O03E1OC4KS2H58SKBN55B60$
   $HT68DBF61BPGAMCNMBE1NVO3PJVQT85J1KCTNVTVLP7B9$
   $F21PFQ80GMN2FDFGSRC$

$P_{\text{candidato}}[x]$	Condições Fronteira
$P_{\text{candidato}}[0]$	[0, 0, 1, 1]
$P_{\text{candidato}}[1]$	[1, 0, 1, 1]
$P_{\text{candidato}}[2]$	[0, 0, 1, 0]
$P_{\text{candidato}}[3]$	[0, 1, 1, 0]
$P_{\text{candidato}}[4]$	[1, 1, 0, 1]
$P_{\text{candidato}}[5]$	[1, 0, 0, 0]
$P_{\text{candidato}}[6]$	[1, 1, 1, 0]
$P_{\text{candidato}}[7]$	[1, 1, 1, 1]
$P_{\text{candidato}}[8]$	[1, 0, 1, 0]
$P_{\text{candidato}}[9]$	[0, 0, 1, 1]
$P_{\text{candidato}}[10]$	[0, 1, 0, 1]

Tabela 4.30: Condições de Fronteira cada $P_{candidato}[x]$

5. $P_{candidatos}[4] = U774V69JAJN3A27O07E1OC4LT0H18GL9BD5B6OHV$
$Q9DBF60RHGAMCNMBF0NFO3HIUQT85J0GCLNUTFLP7R9FE$
$9DD24GGIN2FFEGORT$

6. $P_{candidatos}[5] = MN6KVN9HA773327GGLO1OC4LE0H1RCNV2S4D6$
$OHTM0DB76BAVGAM4NM9U0N3Q2OJUQ9GDJ62DLLUT7LR7T9$
$FE997A5JGIQ6ECF0GRT$

7. $P_{candidatos}[6] = U770V69IAJN3A2FO07E5IC6LT0J0CGL9BF5BMOH$
$VQ9DBE60R0GRICNMBF0NFO31IEQT85J0GCLMUTENP7R9FU$
$99D64GG2N2FFEOORT$

8. $P_{candidatos}[7] = MN74V69HANN3227G0NC1OC4LU0H1B4NR3T5B6O$
$HTU0DB768RHGAMCNM9E0N7Q3PIUQ98DJ2ICLLUT7LP7T9F$
$E99F25GGIR6FEEGORT$

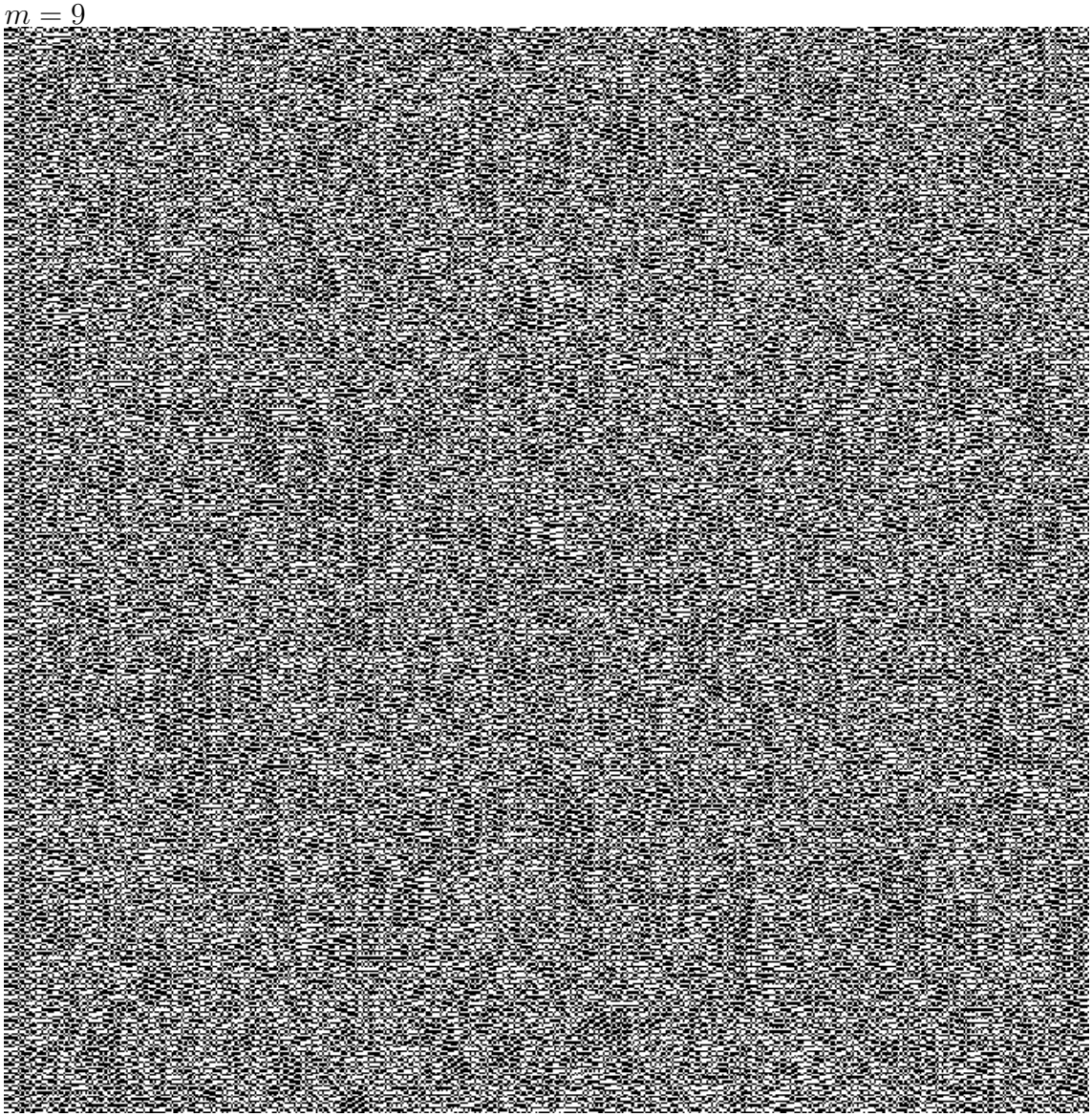
9. $P_{candidatos}[8] = MN6KV69HA7N332FG0LS1OC4LU0H1RCNR2S496O$
$HTM0DB769RLGAMCNM9U0N3Q3PJUQ90DJ2ICLLUT7LP7T9F$
$E997A5HGIR6EEF0ORT$

10. $P_{candidatos}[9] = ML6KV49IAJN332FO0JS4QC45T8J1R4LR2C52MOH$
$TM9DBE69VKGAMCNMBV073Q31JUQD85N2GCLLUTFLT7B9F$
$U997A50GIR6FCMGPRO$

11. $P_{candidatos}[10] = ML71V69IBNN3227O0JE5IC6LTSH084L9BT5BMQ$
$HTU4DB660VGGBICNMBV07FO3HISQ985J0ICLLUT7NT7P9FE$
$19D64GGIJ67FUIPRP$

De todos os elementos criados nesta execução existe apenas 1 que entrará, pelas mesmas razões que as anteriores, na população ideal final, tratando-se do elemento $P_{candidato}[1]$, como podemos ver na figura 4.16, que entrará na população ideal final.

Figura 4.16: Gráfico da Segunda *Code Rule*, $P_{candidato}[1]$, da Quinta Execução para $m = 9$



4.3 Resultados para $m = 15$

Para $m = 15$ os resultados foram inesperadamente maus. Foram realizadas diversas execuções do algoritmo com m=15 mas, no entanto, nunca foi possível chegar a resultados, pois, nenhum conseguiu passar aos testes explicados acima, das frequências.

Como referido anteriormente, o algoritmo principal decorre durante 1000 segundos, com 100 condições iniciais e, quando nenhuma das 15 *code rules* gerou resultados que passaram aos 2 testes necessários, o tempo volta a ser igual a 0, uma nova

população inicial é gerada e é realizado o processo todo novamente.

Existiram várias tentativas de execução com $m = 15$, cada uma com um período de tempo longo, e o programa nunca conseguiu obter resultados interessantes. Assim, percebemos que a premissa sobre existirem melhores resultados quanto maior fosse o valor de m, falada na secção sobre conceitos, é na verdade falsa após ver os resultados.

Na próxima secção de discussão de resultados, será discutido o porquê do facto de se ter melhores resultados com um m menor.

Capítulo 5

Discussão de Resultados

Após a apresentação dos resultados, pode-se passar para a análise dos mesmos. Ao explorar e interpretar os dados obtidos, busca-se uma compreensão mais abrangente das implicações e tendências observadas. Nesta discussão, serão examinados os padrões identificados, as perceções relevantes e as conexões com os objetivos de pesquisa, com o intuito de contribuir para uma visão mais esclarecedora do contexto em questão. Será também explorado um dos resultados mais interessantes - o porquê de quando temos um m maior, obtermos resultados piores.

Em primeiro lugar, vale recapitular quais seriam os resultados esperados e o objetivo desta tese.

5.1 Objetivo

O objetivo desta tese passava por criar um algoritmo genético que criasse de forma simplificada *code rules* que poderiam ser candidatas a serem um **PRNG**. Como tal, foi criado o algoritmo principal já explicado e foram criadas diversas *code rules*, mas apenas aquelas que passavam aos testes de frequências puderam ser consideradas como possíveis candidatas a **PRNG**. Após a criação e feitos os testes em cada uma das *code rules* os gráficos foram visualizados e apresentados. Dos elementos que passaram os testes, apenas aqueles que apresentavam aleatoriedade nos gráficos, através da visualização do seu fenótipo, foram consideradas candidatas a **PRNG**.

No fim, queríamos ter uma população denominada como população ideal final que contivesse todos os elementos que foram considerados candidatos. Foram realizados testes com diferentes valores para m para verificar se a premissa que dizia que o esperado era que quanto maior fosse este valor, melhores seriam os resultados é, ou não, verdadeira.

Como tal, nas próximas sub-secções serão analisados os resultados obtidos através das execuções para cada um dos valores utilizados para m e, no fim, será realizada

uma discussão sobre a premissa mencionada. Em cada um dos resultados, foi realizada uma divisão dos melhores e piores resultados e, inicialmente, o objetivo era apenas adicionar à população ideal final os elementos presentes nas melhores metades que apresentavam gráficos aparentemente aleatórios. No entanto, a discrepância dos resultados dos testes de frequências, como vimos nas tabelas em cada uma das execuções para cada m como por exemplo na tabela 4.1, não é significativa e, algumas vezes, os gráficos da pior metade apresentavam maiores índices de aleatoriedade que a melhor metade. Podemos, então, concluir que como não existe uma discrepância significativa dos resultados dos testes, não faz sentido descartar as piores metades.

5.2 Análise dos Resultados para $m = 7$

As execuções realizadas para $m = 7$ apresentaram diversos elementos que foram considerados relevantes e que foram integrados na população ideal final, sendo este o m que, surpreendentemente, apresentou os melhores e o maior número de resultados. É importante reforçar que cada um dos elementos que entraram na população passaram aos testes das frequências e do desvio padrão.

5.2.1 Primeira Execução

A primeira execução foi considerada um sucesso total, onde todos os 8 elementos foram adicionados à população, uma vez que não só passaram aos testes, como podemos ver na tabela 4.1 e na tabela 4.5, como também apresentaram um gráfico com aparência aleatória.

Este é o primeiro resultado apresentado e começamos a perceber que, talvez, a premissa mencionada seja falsa. No entanto, poderá ter existido alguma sorte nas *code rules* criadas e talvez tivesse sido um caso único.

Concluindo, a população ideal final terá, neste momento, todos os elementos resultantes desta execução. De modo a ser fácil a representação da população ideal final e dos elementos que entraram na mesma de cada execução, seja a primeira $P_{idealfinal}$ e a segunda = $P_{candidatos\_m=7.1}$, onde o ".1" representa o número da execução. Sendo assim, todos os elementos de $P_{candidatos\_m=7.1}$ são adicionados a $P_{idealfinal}$, tendo um tamanho de 8.

5.2.2 Segunda Execução

Nesta execução, os resultados foram quase opostos à anterior, tendo apenas um elemento que foi considerado relevante para entrada na população ideal final, o $P_{candidatos\_m=7.2}[0]$, uma vez que foi o único que não apresentou periodicidade nem estruturas. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.4 e 4.5.

Como tal, a $P_{idealfinal}$ recebe este elemento e passa a ter um tamanho de 9.

5.2.3 Terceira Execução

Na terceira execução com $m = 7$, foram considerados para entrar 2 dos 3 elementos gerados, o $P_{candidatos\_m=7.3}[1]$ e $P_{candidatos\_m=7.3}[2]$. A população ideal final passa a ter um tamanho igual a 11 e os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.7 e 4.8.

5.2.4 Quarta Execução

A penúltima execução com $m = 7$, tal como a primeira, obteve um sucesso total. Todos os 8 elementos obtidos através desta execução tiveram acesso direto à $P_{idealfinal}$. Novamente, estes resultados começam a apresentar um padrão, onde em cada uma das execuções existe novas entradas de forma consistente, ao contrário dos próximos m, onde existem casos onde não entraram nenhum dos elementos, e, para além disso, duas das execuções apresentaram elementos que tiveram entrada direta na população ideal final. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.10 e 4.11. Assim, a população ideal final terá agora um tamanho igual a 19.

5.2.5 Quinta Execução

Por fim, a última execução obteve também 8 elementos e, novamente, foi feita uma adição de 4 elementos à população ideal final, mostrando que, de facto, os resultados com $m = 7$ mantiveram-se consistentemente bons nas 5 execuções. Os elementos que darão entrada na $P_{idealfinal}$ serão os elementos $P_{candidatos\_m=7.5}[0]$, $P_{candidatos\_m=7.5}[1]$, $P_{candidatos\_m=7.5}[4]$ e $P_{candidatos\_m=7.5}[6]$. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.13 e 4.14.

Concluímos, assim, a análise dos elementos gerados em cada uma das execuções de $m = 7$. Ficando com uma $P_{idealfinal}$ com um tamanho de 23. É de frisar que este foi um resultado muito positivo e que, como vamos ver, não será novamente atingido por nenhum dos próximos m.

5.3 Análise dos Resultados para $m = 9$

Agora que já estudámos os resultados de $m = 7$, podemos avançar para o estudo dos resultados para $m = 9$. Como vamos perceber, estes testes resultaram em muitos elementos escolhidos para entrar na população e existe inconsistência de entrada de novos elementos. No início desta análise, a $P_{idealfinal}$ conta com 23 elementos, todos vindos do m anterior.

5.3.1 Primeira Execução

Esta execução deu origem a 3 possíveis entradas na população ideal final. No entanto, podemos perceber que os resultados com $m = 9$ apresentam uma falta de

consistência em apresentar bons resultados, uma vez que nenhum dos elementos gerados foi considerado relevante e, consequentemente, não vai existir nenhuma entrada na população ideal final. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.16 e 4.17.

5.3.2 Segunda Execução

Esta execução gerou bastantes novas possíveis entradas na população ideal final, tendo sido ao todo 11. Inicialmente, parecia que a não entrada de nenhum elemento na população ideal final na última execução poderia ter sido má sorte mas apenas 1 elemento dos 11 gerados foi considerado aleatório o suficiente, através do gráfico, para ser integrado na $P_{idealfinal}$, que se tratou do elemento $P_{candidatos\_m=9.2}[1]$. Assim, a $P_{idealfinal}$ conta agora com 24 elementos. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.19 e 4.20.

5.3.3 Terceira Execução

Nesta execução foram novamente gerados 11 elementos e, reforçando a que a premissa é falsa, nenhum dos elementos foi considerado aleatório e, como tal, nenhum entrou na população ideal final. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.22 e 4.23.

Os resultados desta execução foram uma surpresa, uma vez que, à primeira vista, parecem aleatórios mas o lado esquerdo de cada gráfico apresenta muitos padrões ou estruturas.

5.3.4 Quarta Execução

As duas últimas execuções foram cruciais para perceber se os resultados para $m = 9$ são realmente inconsistentes. Nesta quarta execução, foram gerados 6 elementos, onde 2 foram considerados relevantes para entrada na $P_{idealfinal}$: $P_{candidatos\_m=9.4}[2]$ e $P_{candidatos\_m=9.4}[5]$. Este resultado é melhor que os anteriores, no entanto menos metade foram adicionados, reforçando que existe uma inconsistência de bons resultados. Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.25 e 4.26.

Assim, a $P_{idealfinal}$ tem agora o tamanho de 26 elementos.

5.3.5 Quinta Execução

Por fim, chegamos à última execução realizada com $m = 9$. Desta execução, foram originados 11 elementos, onde apenas 1 teve acesso à população ideal final, confirmando a falta de consistência e uma performance pior que $m = 7$ no que toca a gerar resultados. O elemento que entrou na população ideal final foi o $P_{candidatos\_m=9.5}[1]$.

Os resultados do teste de frequências e dos desvios padrão podem ser vistos, respetivamente, nas tabelas 4.28 e 4.29.

Com estes resultados, podemos observar que a performance de $m = 7$ foi significativamente superior ao de $m = 9$, enquanto que em $m = 7$ foram obtidos 23 elementos para entrar na população ideal final, os resultados desta subsecção apenas deram origem a 4 elementos, uma enorme diferença.

A diferença entre a utilização de um m menor e a utilização de um m maior será ainda mais significativa na apresentação dos resultados da próxima secção.

5.4 Análise dos Resultados para $m = 15$

Seria de esperar que as execuções com $m = 15$ trouxessem resultados muito melhores que os restantes, no entanto, foi precisamente o contrário. Não foi possível gerar um único resultado que pudesse entrar na população ideal final isto, pois, em nenhuma das tentativas realizadas foram geradas *code rules* que passassem aos testes de frequências e desvios padrão, tendo sempre valores de $\delta \geq 0.05$ e $\delta_2 \geq 0.04$, respetivamente.

Como já referido, sempre que o programa não encontra *code rules* da população inicial que passem aos testes, é tudo gerado novamente e o tempo começa a contar novamente. Como tal, o programa foi deixado a executar durante diversas horas e nunca foi possível obter uma única *code rule* que pudesse fazer parte da população ideal final, nem resultados relevantes, uma vez que todos apresentavam médias maiores que os valores de δ e desvios padrão maiores que δ_2. Assim, a $P_{idealfinal}$ acabou com 27 elementos.

5.5 Implicações na Escolha do Tamanho de m e Considerações Finais

Nesta fase final da análise dos resultados, emergem conclusões que divergem das expectativas iniciais. A teoria sustentava a noção de que um valor mais elevado para o parâmetro m resultaria numa maior diversidade e qualidade dos autómatos gerados. Contudo, os resultados obtidos revelaram uma tendência contrária, com um desempenho mais significativo para valores menores de m, entre os considerados.

Ao analisar o caso específico de $m = 15$, surge uma conclusão intrigante. A ausência de resultados satisfatórios pode ser justificada pelo crescimento exponencial do espaço de possíveis regras, o que dificulta a identificação de regras que cumpram os critérios definidos. Tais regras, que seguramente existem algumas, serão potencialmente de melhor qualidade.

A análise dos resultados para $m = 7$ e $m = 9$ merece destaque, especialmente no primeiro caso. A existência de resultados promissores com $m = 7$ evidencia

a capacidade de gerar autómatos com potencial para serem utilizados como um **PRNG**. A menor quantidade de resultados satisfatórios para $m = 9$ reflete a maior dificuldade em encontrar regras que se enquadrem nos critérios estabelecidos, para valores de m maiores.

A consideração do tamanho da amostra de teste revela-se crucial neste contexto. A amostra mais extensa nos casos de $m = 7$ aumenta a probabilidade de identificar elementos de elevada qualidade, permitindo a geração de múltiplos candidatos promissores.

As conclusões obtidas realçam a importância de considerar não apenas a dimensão do espaço de busca, mas também a viabilidade de explorá-lo eficientemente. Resumindo, os resultados apresentados expõem um cenário desafiador e surpreendente, onde a relação entre o valor de m e a qualidade dos resultados não segue o padrão intuitivo. As considerações finais refletem a necessidade de abordagens adaptativas e exploratórias ao enfrentar problemas complexos de otimização, refletindo sobre as lições aprendidas durante o estudo.

5.6 Conclusão da Análise de Resultados

Recapitulando, o objetivo do presente estudo era criar uma algoritmo generativo com capacidade de criar *code rules* que pudessem ser consideradas candidatas a ser um **PRNG**.

Para isso tinham que ter os critérios previamente definidos e apresentar uma estrutura aparentemente aleatória, com isenção de periodicidade e de estruturas.

Assim, o algoritmo generativo realizado, numa execução com população inicial de 15 elementos gerados aleatoriamente com todas as características descritas anteriormente, foi capaz de criar 27 elementos candidatos a **PRNG**. A população ideal final que contém os resultados do algoritmo generativo para as condições iniciais especificadas é a seguinte:

$$P_{idealfinal} =$$
$$\{P_{candidatos\_m=7.1}[0], P_{candidatos\_m=7.1}[1], P_{candidatos\_m=7.1}[2], P_{candidatos\_m=7.1}[3],$$
$$P_{candidatos\_m=7.1}[4], P_{candidatos\_m=7.1}[5], P_{candidatos\_m=7.1}[6], P_{candidatos\_m=7.1}[7],$$
$$P_{candidatos\_m=7.2}[0], P_{candidatos\_m=7.3}[1], P_{candidatos\_m=7.3}[2], P_{candidatos\_m=7.4}[0],$$
$$P_{candidatos\_m=7.4}[1], P_{candidatos\_m=7.4}[2], P_{candidatos\_m=7.4}[3], P_{candidatos\_m=7.4}[4],$$
$$P_{candidatos\_m=7.4}[5], P_{candidatos\_m=7.4}[6], P_{candidatos\_m=7.4}[7], P_{candidatos\_m=7.5}[0],$$
$$P_{candidatos\_m=7.5}[1], P_{candidatos\_m=7.5}[4], P_{candidatos\_m=7.5}[6], P_{candidatos\_m=9.2}[1],$$
$$P_{candidatos\_m=9.4}[2], P_{candidatos\_m=9.4}[5], P_{candidatos\_m=9.5}[1]\}$$

Como tal, considera-se que o objetivo da tese foi cumprido e que é possível utilizar autómatos para criar candidatos a **PRNG**. Apesar disso, existem limitações, como o facto de estas 27 *code rules* serem apenas candidatos a **PRNG**. No entanto, o objetivo é criar de forma simples e rápida autómatos celulares candidatos a **PRNG** e, como resultados, foram obtidos 27.

Capítulo 6

Conclusões e trabalho futuro

No decurso deste estudo, foram apresentados os conceitos fundamentais necessários para compreender o funcionamento do algoritmo generativo que foi desenvolvido, bem como a sua aplicabilidade. O objetivo primordial era criar um algoritmo capaz de produzir candidatos a **PRNG** de forma eficiente e eficaz. Conclui-se, com satisfação, que este objetivo foi alcançado, com o algoritmo a gerar 27 candidatos a **PRNG**, após uma série de execuções sobre as condições especificadas.

Este trabalho representa um marco no caminho para a criação de novos geradores de números pseudo-aleatórios (**PRNGs**) através de abordagens generativas. Apesar das questões e incertezas associadas à validação final destes resultados, o presente estudo abre a possibilidade de gerar candidatos que podem ser posteriormente estudados e, eventualmente, incorporados em sistemas com facilidade e rapidez.

A relevância deste estudo no contexto mais amplo da geração de números pseudo-aleatórios, reside na introdução de uma abordagem generativa, que tem como propósito diversificar os **PRNGs** disponíveis e, assim, contribuir para uma maior resiliência e segurança nas aplicações que dependam de números aleatórios.

Como referido anteriormente, o próximo passo envolverá a realização de testes rigorosos, como os testes **NIST** previamente delineados, para avaliar a qualidade e confiabilidade dos 27 candidatos a **PRNG** gerados. Esta fase de testes revelará quais destes candidatos podem realmente ser considerados **PRNGs** robustos e adequados para aplicações reais.

Neste sentido, o presente estudo não representa o fim da jornada, mas sim um passo fundamental na direção da inovação e aprimoramento da geração de números pseudo-aleatórios. Este trabalho tem potencial para ter um impacto no estudo de geradores de números pseudo-aleatórios introduzindo uma abordagem generativa que promete diversidade e inovação, com possíveis aplicações em áreas tão diversas como segurança de dados, criptografia e simulações computacionais. O trabalho futuro será crucial para validar e implementar efetivamente os **PRNGs** gerados, dando continuidade a esta pesquisa.

Concluindo, os resultados demonstraram que, de facto, a utilização deste algoritmo generativo criou 27 candidatos, ou *code rules*, relevantes que podem ser um **PRNG**, completando o objetivo do estudo. No entanto, existem limitações, como o facto de estas 27 *code rules* serem apenas candidatas a **PRNG** e, também, do facto de alguns dos candidatos que foram rejeitados por causa de terem, visualmente, padrões ou estruturas poderem ser considerados um **PRNG** pelos testes **NIST**, uma vez que os fenótipos serão analisados de forma computacional.

Bibliografia

Adamatzky, Andrew (2010). *Game of life cellular automata*. Vol. 1. Springer.

Afonso, Anabela e Carla Nunes (2019). *Probabilidades e Estatística. Aplicações e Soluções em SPSS. Versão revista e aumentada.*

Andrey (dez. de 2022). *What the hack is Rule 30? Cellular Automata Explained*. URL: https://gettocode.com/2021/10/17/what-the-hack-is-rule-30-cellular-automat-explained/.

Arun, Swathi (jan. de 2022). *The Secrets Module of Python - The Pythoneers*. URL: https://medium.com/pythoneers/the-secrets-module-of-python-150af4c9f300.

Bassham III, Lawrence E et al. (2010). *Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology.

Bellare, Mihir, Shafi Goldwasser e Daniele Micciancio (1997). ""Pseudo-random" number generation within cryptographic algorithms: The DDS case". Em: *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*. Springer, pp. 277–291.

Bhattacharjee, Kamalika e Sukanta Das (2022). "A search for good pseudo-random number generators: Survey and empirical studies". Em: *Computer Science Review* 45, p. 100471.

Boccaletti, Stefanos et al. (2000). "The control of chaos: theory and applications". Em: *Physics reports* 329.3, pp. 103–197.

Carlos Ramos, Marta Riera (2009). "Evolutionary dynamics and the generation of cellular automata". Em: *Iteration theory (ECIT '08), 219-236, Grazer Math.Ber., 354, Institut für Mathematik, Karl-Franzens-Universität Graz, Graz.*

Dar, Reuven, Ronald C Serlin e Haim Omer (1994). "Misuse of statistical tests in three decades of psychotherapy research." Em: *Journal of consulting and clinical psychology* 62.1, p. 75.

DiCarlo, David F (2012). "Random number generation: Types and techniques". Em.

Erlandsson, Jonas Schubert (s.d.). *Duty calls - CSS3 is NOT proven to be turing complete!* URL: https://accodeing.com/blog/2015/duty-calls-css3-is-not-proven-to-be-turing-complete.

Gage, Dustin, Elizabeth Laub e Briana McGarry (2005). "Cellular automata: is rule 30 random". Em: *Proceedings of the Midwest NKS Conference, Indiana University*.

Goldwasser, Shafi e Mihir Bellare (1996). "Lecture notes on cryptography". Em: *Summer course "Cryptography and computer security" at MIT* 1999, p. 1999.

IBM (s.d.). *What is Monte Carlo Simulation?* URL: `https://www.ibm.com/topics/monte-carlo-simulation`.

Jagannatam, Archana (2008). "Mersenne Twister–A Pseudo random number generator and its variants". Em: *George Mason University, Department of Electrical and Computer Engineering*.

Kunkle, Daniel R (2003). "Automatic classification of one-dimensional cellular automata". Tese de doutoramento. Rochester Institute of Technology.

L'Ecuyer, Pierre (1992). *Testing random number generators*. Rel. téc. Institute of Electrical e Electronics Engineers (IEEE).

Li, Chung-Chih e Bo Sun (2005). "Using Linear Congruential Generators for Cryptographic Purposes." Em: *CATA*, pp. 13–19.

Mahto, Dindayal, Danish Ali Khan e Dilip Kumar Yadav (2016). "Security analysis of elliptic curve cryptography and RSA". Em: *Proceedings of the world congress on engineering*. Vol. 1, pp. 419–422.

Marton, Kinga, Alin Suciu e Iosif Ignat (2010). "Randomness in digital cryptography: A survey". Em: *Rom. J. Inf. Sci. Technol* 13.3, pp. 219–240.

Mattioli, Federico (2019). "Testing a Random Number Generator: formal properties and automotive application." Em: *Master's thesis, Alma Mater Studiorum - Università di Bologna*.

Menezes, Alfred J, Paul C Van Oorschot e Scott A Vanstone (2018). *Handbook of applied cryptography*. CRC press.

Mengdi, Zhang et al. (2021). "Overview of Randomness Test on Cryptographic Algorithms". Em: *Journal of Physics: Conference Series*. Vol. 1861. 1. IOP Publishing, p. 012009.

Mitchell, Melanie (1995). "Genetic algorithms: An overview." Em: *Complex*. Vol. 1. 1. Citeseer, pp. 31–39.

Naik, Rasika B e Udayprakash Singh (2022). "A Review on Applications of Chaotic Maps in Pseudo-Random Number Generators and Encryption". Em: *Annals of Data Science*, pp. 1–26.

Obe, Bill Buchanan (dez. de 2021). *RSA and Random Bitstream Generators - Prof Bill Buchanan OBE*. URL: `https://billatnapier.medium.com/rsa-and-random-bitstream-generators-532e857ac890`.

PEP 506 – Adding A Secrets Module To The Standard Library | peps.python.org (s.d.). URL: `https://peps.python.org/pep-0506/`.

Raeside, DE (1976). "Monte Carlo principles and applications". Em: *Physics in Medicine & Biology* 21.2, p. 181.

Ramos, Carlos, Fernando Carapau e Paulo Correia (2022). "Cellular Automata Describing Non-equilibrium Fluids with Non-mixing Substances". Em: *Recent Advan-*

ces in Mechanics and Fluid-Structure Interaction with Applications: The Bong Jae Chung Memorial Volume. Springer, pp. 229–245.

random — Generate pseudo-random numbers — Python 3.11.0 documentation (s.d.). URL: https://docs.python.org/3/library/random.html.

Ritter, Terry (1991). "The efficient generation of cryptographic confusion sequences". Em: *Cryptologia* 15.2, pp. 81–139.

Shiffman, Daniel (s.d.). *The Nature of Code*. URL: https://natureofcode.com/book/chapter-7-cellular-automata/.

Sinha, Vaarun (nov. de 2021). *Why you should never use random module for generating passwords.* URL: https://dev.to/vaarun_sinha/why-you-should-never-use-random-module-for-generating-passwords-38nl.

Tomassini, Marco e Mathieu Perrenoud (2001). "Cryptography with cellular automata". Em: *Applied Soft Computing* 1.2, pp. 151–160. ISSN: 1568-4946. DOI: https://doi.org/10.1016/S1568-4946(01)00015-1. URL: https://www.sciencedirect.com/science/article/pii/S1568494601000151.

Turney, Shaun (out. de 2022). *Chi-square goodness of fit test: Formula, Guide & Examples*. URL: https://www.scribbr.com/statistics/chi-square-goodness-of-fit/.

, TylerMSFT (out. de 2022). *rand*. URL: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/rand?view=msvc-170.

Wang, Luyao e Hai Cheng (2019). "Pseudo-random number generator based on logistic chaotic system". Em: *Entropy* 21.10, p. 960.

Weisstein, Eric W. (s.d.[a]). *"Rule 110."From MathWorld–A Wolfram Web Resource.* URL: https://mathworld.wolfram.com/Rule110.html.

— (s.d.[b]). *"Rule 30."From MathWorld–A Wolfram Web Resource.* URL: https://mathworld.wolfram.com/Rule30.html.

Wolfram, Stephen (1983). "Cellular automata". Em: *Los Alamos Science, http://library. lanl. gov/cgi-bin/getfile*, pp. 09–01.

— (1986). "Cryptography with cellular automata". Em: *Advances in Cryptology—CRYPTO'85 Proceedings 5*. Springer, pp. 429–432.

— (2002). *A New Kind of Science*. English. Wolfram Media. ISBN: 1579550088. URL: https://www.wolframscience.com.